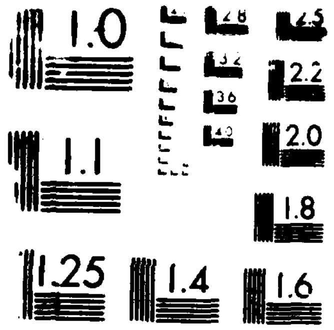
AD-A188 617 TWO PAPERS ON A SYMBOLIC ANALYZER FOR MOS (METAL-OXIDE SEMICONDUCTORS) CI (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE R E BRYANT DEC 87 1/1

UNCLASSIFIED CMU-CS-87-106 AFWAL-TR-87-1167 F/G 12/2 NL



MICROCOPY RESOLUTION TEST CHART

PHOTOGRAPH THIS SHEET

AD-A188 617

DTIC ACCESSION NUMBER

LEVEL

INVENTORY 1

AFWAL-TR-87-1167

DOCUMENT IDENTIFICATION

Dec 1987

This document has been approved for public release and sale; its distribution is unlimited.

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS GRA&I	☒
DTIC TAB	☐
UNANNOUNCED	☐
JUSTIFICATION	
BY	
DISTRIBUTION	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A-1	

DISTRIBUTION STAMP

QUALITY INSPECTED 2

DTIC SELECTED FEB 0 9 1988 E

DATE ACCESSIONED

DATE RETURNED

88 2 05 101

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

DTIC FORM 70A DEC 83

DOCUMENT PROCESSING SHEET

PREVIOUS EDITION MAY BE USED UNTIL STOCK IS EXHAUSTED.

AD-A188 617

AFWAL-TR-87-1167

TWO PAPERS ON A SYMBOLIC ANALYZER FOR MOS CIRCUITS

R.E. Bryant

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim

Approved for Public Release; Distribution is Unlimited

AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Chahira M. Hopper

CHAHIRA M. HOPPER
Project Engineer

Richard C Jones

RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER

Edward L. Gliatti

EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE — Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b DECLASSIFICATION DOWNGRADING SCHEDULE	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
4 PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-87-106	5 MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1167
6a NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories AFWAL/AAAT-3
6c ADDRESS (City, State, and ZIP Code) Computer Science Dept Pittsburgh PA 15213-3890	7b ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543
8a NAME OF FUNDING SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-84-K-1520
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO 61101E; PROJECT NO 4976; TASK NO 00; WORK UNIT ACCESSION NO 01

11 TITLE (Include Security Classification)
Two Papers on a Symbolic Analyzer for MOS Circuits

12 PERSONAL AUTHOR(S)
R. E. Bryant

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Interim	FROM ____ TO ____	1987 December	37

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	switch networks; symbolic analysis; MOS circuit analysis; Gaussian elimination; series-parallel graphs; Boolean manipulation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

A network of switches controlled by Boolean variables can be represented as a system of Boolean equations. The solution of this system gives a symbolic description of the conducting paths in the network. Gaussian elimination provides an efficient technique for solving sparse systems of Boolean equations. For the class of networks that arise when analyzing digital metal-oxide semiconductor (MOS) circuits, a simple pivot selection rule guarantees that most s switch networks encountered in practice can be solved with $O(s)$ operations. When represented by a directed acyclic graph, the set of Boolean formulas generated by the analysis has total size bounded by the number of operations required by the Gaussian elimination. This paper presents the mathematical basis for systems of Boolean equations, their solution by Gaussian elimination, and data structures and algorithms for representing and manipulating Boolean formulas.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Chahira M. Hopper	22b TELEPHONE (Include Area Code) (513) 255-7865 — 22c OFFICE SYMBOL AFWAL/AAAT-3

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

SUMMARY

This report contains two papers describing a set of algorithms to extract the logical behavior of a digital metal-oxide semiconductor (MOS) from its transistor representation. Switch-level network analysis, applied symbolically, performs the extraction. The analyzer captures all aspects of switch-level networks including bidirectional transistors, stored charge, different signal strengths, and indeterminate (X) logic values. The output is a set of Boolean formulas, where the behavior of each network node is represented by a pair of formulas. In the worst case, the analysis of an n node network can yield a set of formulas containing a total of $O(n^3)$ Boolean operations. However, all but a limited set of dense, pass transistor networks give formulas with $O(n)$ total operations.

The analyzer can serve as the basis of efficient programs for a variety of logic design tasks, including: logic simulation (on both conventional and special purpose computers), fault simulation, test generation, and symbolic verification.

These papers have been accepted for publication in *IEEE Transactions on Computer-Aided Design of Integrated Circuits.*

Algorithmic Aspects of Symbolic Switch Network Analysis*

Randal E. Bryant
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

February 5, 1987

Abstract

A network of switches controlled by Boolean variables can be represented as a system of Boolean equations. The solution of this system gives a symbolic description of the conducting paths in the network. Gaussian elimination provides an efficient technique for solving sparse systems of Boolean equations. For the class of networks that arise when analyzing digital metal-oxide semiconductor (MOS) circuits, a simple pivot selection rule guarantees that most s switch networks encountered in practice can be solved with $O(s)$ operations. When represented by a directed acyclic graph, the set of Boolean formulas generated by the analysis has total size bounded by the number of operations required by the Gaussian elimination. This paper presents the mathematical basis for systems of Boolean equations, their solution by Gaussian elimination, and data structures and algorithms for representing and manipulating Boolean formulas.

Keywords and phrases: switch networks, symbolic analysis, MOS circuit analysis, Gaussian elimination, series-parallel graphs, Boolean manipulation.

1 Introduction

The advent of metal-oxide semiconductor (MOS) circuit technology has revived interest in analyzing networks of switches. This field originated when digital circuits were constructed with electromechanical relays. Shannon, in the first application of Boolean algebra to digital systems, developed several techniques for analyzing a switch network symbolically[1]. For a network of switches, each of which is either open or closed depending on the value of

*This research was supported in part by the Defense Advanced Research Projects Agency, ARPA Order Number 4976, and in part by the Semiconductor Research Corporation under Contract 86-01-068.

some Boolean variable, the goal of symbolic analysis is to derive formulas expressing the conditions under which conducting paths will exist between specified pairs of terminals.[1]

A MOS transistor can often be abstracted as a switch—it conditionally forms a connection between its source and drain nodes depending on the voltage on its gate node. Several models of static logic gates in MOS treat transistors as simple switches and define the behavior of a gate in terms of the conditions under which a conducting path is formed from the supply or ground to the gate output[2,3,4]. More complex MOS models take into account such effects as resistance ratios, dynamic memory, and invalid or uninitialized logic values[5,6]. A companion paper [7] shows that even with these more elaborate models, the behavior of a MOS circuit can be determined by analyzing a series of switch networks. Thus the symbolic analysis of switch networks remains as a key problem in deriving an abstract representation of the function computed by a digital circuit.

The demands placed upon symbolic analysis have changed greatly since the days of relay circuits. These circuits were relatively small, and the analysis was performed manually. Under these conditions, the asymptotic performance of the method matters less than its conceptual simplicity. For example, Shannon describes a method that involves enumerating every possible simple path between two terminals, forming the Boolean product of the switch labels in each path, and then forming the Boolean sum of these path formulas. In general, the number of simple paths in a network can grow exponentially with the number of switches (e.g., the parity ladder shown in Figure 7.) Consequently, path analysis cannot be applied to networks of significant size. Furthermore, there was no concern about data structures and algorithms for representing and manipulating Boolean formulas. Even more recent methods based on matrix representations[8] do not address these algorithmic issues.

Today symbolic analysis methods are to be executed by computers on networks containing thousands of switches. To implement an analyzer, every detail of representation and algorithm must be specified. Success must be measured by worst or average case complexity rather than by performance on small examples. Unfortunately, the state of the art in symbolic analysis has not kept up with these demands. For example, most published symbolic analysis methods for MOS circuits start by enumerating all possible simple paths in the network[9,10,11]. A second method involves enumerating the possible sets of connected components formed in the switch network for different values of the control variables [12]. This approach can also produce a description of size exponential in the number of transistors. These accounts indicate little progress since Shannon's day.

In general, a MOS circuit can be partitioned into smaller subnetworks and each subnetwork analyzed separately. Most of these subnetworks are small—containing no more than 10 transistors. Hence, one can argue that even an algorithm of exponential complexity can work well in practice. However, subnetworks containing over 1000 transistors commonly occur in large pass transistor and datapath circuits. A program for general use cannot rely so heavily on a particular circuit style to achieve tolerable efficiency.

This paper proposes a far more exacting standard for symbolic analysis: that the size

[1]Shannon characterised a network by its "hindrance" function, with logic value 1 indicating the absence of any path. This paper adopts the more conventional view of 1 indicating the presence of a path.

of the symbolic description should be comparable to the size of the original network. That is, a network of s switches should be represented by a set of formulas containing, in total, $O(s)$ Boolean operations. This paper shows how to achieve this goal for most networks arising in the analysis of MOS circuits. Even for the dense pass transistor circuits that lead to a nonlinear complexity, the method produces a description of size $O(s^{3/2})$. This performance results from a combination of several techniques:

- A network is represented by a system of Boolean equations. This system expresses the effects of all paths in the network but lends itself to solution methods of polynomial complexity.

- The system of equation is solved symbolically by Gaussian elimination. A simple heuristic for selecting pivots guarantees that most practical networks of s switches can be solved with $O(s)$ algebraic operations.

- The set of Boolean formulas is represented by a directed acyclic graph (DAG), with each DAG node specifying a Boolean operation to be applied to its children, and with each leaf denoting a variable or constant. This representation naturally allows sharing of common subexpressions. The number of nodes in the DAG is bounded by the number of algebraic operations required during Gaussian elimination.

- Expression simplification techniques are applied to the DAG but only in ways that reduce the size of the DAG even further.

The algorithm presented is efficient in terms of execution speed as well as the size of the result produced.

The formulation of the switch network analysis problem used in this paper is tailored to the particular needs of the MOS circuit analysis method presented in the companion paper. It differs from the classic formulation in the following respects:

- The control signals of switches can be arbitrary Boolean formulas, not just variables or their complements.

- Switches are directed—the conduction conditions from one point to another can differ from those in the reverse direction.

- The analysis does not compute the conduction conditions between specified pairs of terminals. Instead, each node is given an initial value represented by a Boolean formula. A path is viewed as having an "effect" on its destination node equal to the Boolean product of formulas representing the initial value on the source node and the control signals of the switches. Symbolic analysis derives a formula for each node describing the Boolean sum of the effects of all paths to the node.

- The analysis may need to characterize the absence rather than the presence of conducting paths.

Each of these differences represents only a slight generalization of the original problem without increasing its complexity. The requirement for directed switches may seem counterintuitive given that MOS transistors are fully bidirectional devices. However, the MOS analyzer accounts for signals of varying strength (representing different driving admittances) by assigning different labels to the two directed edges representing a transistor.

This paper presents some new results on the efficiency of Gaussian elimination for solving systems of equations defined over general series-parallel graphs. Otherwise, it contains little that has not been presented in some form elsewhere. However, material has been drawn from a diversity of disciplines, including switching theory, graph theory and algorithms, linear systems, optimizing compilers, and symbolic manipulation. In many cases, ideas or techniques are applied in ways much different from those conceived by their developers. Few practitioners in the field of computer-aided design are well versed in all of these disciplines. Furthermore, other presentations of methods for solving path problems in graphs have been in terms far more general, abstract, and harder to understand. The main contribution of this paper is to synthesize a collection of ideas into a single framework for solving an important problem.

Sections 2–4 present a mathematical description of the symbolic analysis problem in terms of systems of equations defined over a Boolean algebra. It parallels previous work on the symbolic analysis of contact networks [8], and more general algebraic formulations of path problems [13,14,15,16]. The presentation differs from previous ones in several respects. First, Boolean algebra is selected as the domain of interest. This gives properties that more general presentations cannot assume, including a finite, partially ordered domain, and an idempotent sum operation. In addition, systems of equations are expressed in terms of labeled graphs rather than with matrix algebras. Graphs more clearly capture the sparse structure of the problem to be solved and directly map into data structures for efficient algorithms. Section 5 describes how a system of Boolean equations can be solved by Gaussian elimination. A combination of formal and empirical arguments shows that most networks arising when analyzing MOS circuits can be solved with a linear number of operations using a simple pivot selection rule. This result could prove useful for other circuit analysis programs such as circuit-level simulators. Section 6 describes data structures and algorithms for representing and simplifying Boolean formulas. Section 7 presents an example showing some strengths and weaknesses of the method, and Section 8 summarizes the results.

2 Symbolic Algebra

Symbolic analysis derives formulas that express conditions under which conducting paths are formed in a network of switches. Such formulas are concrete, syntactic representations of Boolean functions. Several formulas may represent a single function. In mathematical terms, the analyzer computes over a domain consisting of the set of all functions mapping a set of p variables (describing the control signals on the switches) to the set $\{0,1\}$, i.e.,

$$\mathcal{B} = \left\{ f\colon \{0,1\}^p \rightarrow \{0,1\} \right\}$$

In the Boolean algebra of the analysis $\langle \mathcal{B}, \wedge, \vee, \neg, \mathbf{0}, \mathbf{1} \rangle$ the operations $\wedge$, $\vee$, and $\neg$ denote Boolean AND, OR, and NOT, respectively, applied to *functions*. The distinguished elements $\mathbf{0}$ and $\mathbf{1}$ denote the constant functions that yield 0 and 1, respectively, for all argument values. This process of *abstracting* from a primitive domain of Boolean values to one of functions over these values forms the basis of symbolic analysis. Most algebraic properties carry over from the original domain to the abstract one.

The Boolean product of the elements in a set A is denoted $\bigwedge_{a \in A} a$. The product of an empty set is defined to equal $\mathbf{1}$. Similarly, the Boolean sum of the elements in a set A is denoted $\bigvee_{a \in A} a$. The sum of an empty set is defined to equal $\mathbf{0}$.

Elements of $\mathcal{B}$ are partially ordered as $b \leq a$ when $b \vee a = a$, i.e., by their lattice ordering [2,17]. This partial ordering obeys the following properties, as can easily be derived from the laws of Boolean algebra:

Proposition 1 *For any $b \in A$*

$$b \leq \bigvee_{a \in A} a$$

Proposition 2 *If $b \geq a$ for all $a \in A$ then*

$$b \geq \bigvee_{a \in A} a.$$

Proposition 3 *If $a \leq b$, then $\neg b \leq \neg a$.*

Proposition 4 *$b \leq a$ if and only if $b \wedge \neg a = \mathbf{0}$.*

3 Systems of Boolean Equations

Just as a resistor network can be represented by a system of linear equations, so a switch network can be represented by a system of equations in which $\vee$ and $\wedge$ replace addition and multiplication, respectively. This section develops the theory of such systems in terms of labeled graphs and then shows how the switch network analysis problem can be formulated in these terms. In this discussion, (V, E) is a finite, directed graph with vertices V and edges $E \subseteq V \times V$, where $|V| = n$.

Definition 1 *A* vertex labeling *x is an assignment $x(v) \in \mathcal{B}$ to each vertex $v \in V$.*

Two vertex labelings x and y are partially ordered $x \leq y$ when $x(v) \leq y(v)$ for all $v \in V$.

Definition 2 *An* edge labeling *A is an assignment $A(u, v) \in \mathcal{B}$ to each edge $(u, v) \in E$.*

Definition 3 *A* system of Boolean equations *$[A, b]$ consists of an edge labeling A and a vertex labeling b.*

Definition 4 *A vertex labeling x* satisfies *the system $[A, b]$ when*

$$x(v) \;=\; b(v) \vee \bigvee_{(u,v)\in E} [x(u) \wedge A(u,v)] \qquad (1)$$

for every $v \in V$.

Observe that unlike the usual formulation of systems of linear equations ($Ax = b$), the unknown x appears on both the left and right hand side of Equation 1. In a matrix notation, this equation would have the form $x = b \vee Ax$. This departure from convention is forced by the fact that the domain has no inverses under $\vee$.

The following property follows directly from the above definition and Proposition 1.

Proposition 5 *If x satisfies a system $[A, b]$, then $x \geq b$.*

In general, many labelings can satisfy a system of equations. For example, any vertex labeling satisfies the system $[I, z]$ defined over a graph with all edges of the form (v, v), where $I(v, v)$ equals **1** and $z(v)$ equals **0** for all v. We focus our attention on a particular one, considered the "solution" of the system.

Definition 5 *Vertex labeling x is a* solution *of the system $[A, b]$, when*

1. *it satisfies the system, and*
2. *$x \leq y$ for any labeling y satisfying the system.*

By this definition, a system can have at most one solution. In fact, we can show that any system has exactly one solution.

Theorem 1 *Any Boolean system $[A, b]$ has a unique solution x given by the limit of the sequence x^i, where $x^0(v) = \mathbf{0}$ for all v, and $x^i(v)$ is defined for all $i > 0$ and all v as:*

$$x^i(v) \;=\; b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]. \qquad (2)$$

A proof of this theorem is given in Appendix A, based on the following series of arguments. First, the sequence satisfies $x^i \leq x^{i+1}$ for all i and therefore converges. Second, the value to which the sequence converges satisfies the system. Finally, for any labeling y satisfying the system, $x^i \leq y$ for all i. Therefore, the sequence converges to the minimum labeling satisfying the system.

A system of Boolean equations defines a path problem as follows. Let $P_{u,v}$ denote the set of paths from vertex u to vertex v. A path contributes an "effect" to its destination vertex equal to the Boolean product of labels on its source vertex and edges. The net effect at a vertex is given by the Boolean sum of the effects of all paths having this vertex as destination. The solution of a Boolean system yields the vertex labeling giving the net effect at each vertex as is expressed in the following theorem.

Theorem 2 *If $x(v)$ is defined for all $v \in V$ as*

$$x(v) \;=\; \bigvee_{u \in V} \bigvee_{p \in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t) \in p} A(s,t) \right]$$

then x is the solution of the system $[A,b]$.

A proof of this theorem is shown in Appendix A based on the following series of arguments. First, x satisfies the system $[A,b]$. Second, for any path p to vertex v and any labeling y satisfying the system, the effect of path p is less than or equal to $y(v)$. Therefore, the combined effects of the paths to every vertex must equal the minimum labeling satisfying the system.

Theorem 2 shows how to formulate the classical switch network analysis problem as one of solving a system of Boolean equations. For example, consider a switch network with a designated source terminal s. Let the graph (V,E) have the nodes of the network as vertices, and have an edge (m,n) for each pair of nodes m and n connected by a switch. Let edge labeling $A(m,n)$ equal the Boolean sum of all signals controlling switches connecting nodes m and n. Define the vertex labeling b as $b(s) = \mathbf{1}$, and $b(n) = \mathbf{0}$ otherwise. Then the solution of the system $[A,b]$ is a vertex labeling x where $x(n)$ describes the conditions under which a conducting path will form from s to node n.

4 Dual Systems

Some applications require formulas expressing conditions for the *absence* of any conducting path between terminals of a switch network. Such formulas could be obtained by first solving a system expressing conditions for the presence of paths and then complementing the solution values. These negation operations, however, complicate the task of simplifying the formulas. Alternatively, the formulas can be derived directly by solving a *dual system* in which the roles of $\vee$ and $\wedge$ are interchanged. The mathematical basis for this technique stems from DeMorgan's Laws.

Definition 6 *A* dual system of Boolean equations $[A,b]^D$ *consists of an edge labeling A and a vertex labeling b.*

Definition 7 *Vertex labeling x* satisfies *the dual system $[A,b]^D$ when*

$$x(v) \;=\; b(v) \wedge \bigwedge_{(u,v) \in E} [x(u) \vee A(u,v)]$$

for every $v \in V$.

As before, only one vertex labeling is considered to solve a dual system, but this time the maximum one is selected.

Definition 8 *Vertex labeling x is a* solution *of the dual system $[A,b]^D$, when*

1. *it satisfies the system, and*
2. *$x \geq y$ for any labeling y satisfying the system.*

Definition 9 *The* complement *of vertex labeling b, denoted $\overline{b}$, is a vertex labeling with $\overline{b}(v) = \neg b(v)$ for all $v \in V$.*

Definition 10 *The* complement *of edge labeling A, denoted $\overline{A}$, is an edge labeling with $\overline{A}(u,v) = \neg A(u,v)$ for all $(u,v) \in E$.*

Theorem 3 *x is the solution of the system $[A,b]$ if and only if $\overline{x}$ is the solution of the dual system $[\overline{A},\overline{b}]^D$.*

The proof of this theorem is given in Appendix A. It involves a straightforward application of DeMorgan's Laws.

From this theorem, the role of dual systems in expressing the absence of paths becomes clear.

Corollary 1 *If x is the solution of the dual system $[\overline{A},\overline{b}]^D$ then*

$$x(v) = \neg \bigvee_{u \in V} \bigvee_{p \in P_{u,v}} b(u) \; [illegible]$$

5 Equation Solution

The equations of the preceding section give a [illegible] function. Symbolic analysis derives explicit [illegible] following presentation describes the solution of [illegible] similarly by interchanging the roles of [illegible] and [illegible]

As Equation 2 suggests, a simple iterative method [illegible] Although this method lacks efficiency, it aids in understanding [illegible] methods. Starting with $x(v) = b(v)$ for each vertex v, the iterative method propagates values from a vertex v through an edge (v,u) to the adjacent vertex u and combines this value with the value already on vertex u. That is, each step involves a computation of the form

$$x(u) \leftarrow x(u) \vee x(v) \wedge A(v,u)$$

Ultimately, this process converges to a solution. However, an iterative method must either test the solution for convergence, i.e., that $x(v) \wedge A(v,u) \leq x(u)$ for all $(v,u) \in E$, or it must perform enough iteration steps to guarantee that for every simple path in the graph, the value on the source vertex has been propagated through the edges of the path to the destination vertex. The first method requires solving the NP-hard problem of testing Boolean formulas for equivalence [18], while the second requires $\Theta(|V| \cdot |E|)$ steps, except for restricted graph structures [19].

```
{ Forward elimination }
$V_0 \leftarrow V$; { The uneliminated vertices }
$E_0 \leftarrow E$; { The original edges plus fill-in's}
for $i \leftarrow 1$ to $n$ do
begin
        choose vertex from $V_{i-1}$ and call it $v_i$; { Select pivot }
        $V_i \leftarrow V_{i-1} - \{v_i\}$;
        $E_i \leftarrow E_{i-1} \cap [V_i \times V_i]$;
        for each $v \in V_i$ such that $(v_i, v) \in E_{i-1}$ do
        begin
                $b(v) \leftarrow b(v) \vee [b(v_i) \wedge A(v_i, v)]$;
                for each $u \in V_i$ such that $(u, v_i) \in E_{i-1}$ and $u \neq v$ do
                begin
                        if $(u, v) \in E_i$
                        then $A(u,v) \leftarrow A(u,v) \vee [A(u,v_i) \wedge A(v_i,v)]$
                        else begin
                                { Create fill-in edge. }
                                $E_i \leftarrow E_i \cup \{(u,v)\}$;
                                $A(u,v) \leftarrow A(u,v_i) \wedge A(v_i,v)$
                        end
                end
        end
end;

{ Back Substitution }
for $i \leftarrow n$ step $-1$ to 1 do
begin
        $x(v_i) \leftarrow b(v_i)$;
        for each $u \in V_i$ such that $(u, v_i) \in E_{i-1}$ do
                $x(v_i) \leftarrow x(v_i) \vee [x(u) \wedge A(u, v_i)]$;
end
```

Figure 1: **Gaussian Elimination Algorithm.** The code differs from the conventional presentation in that it uses graph notation and substitutes Boolean for numerical operations.

5.1 Gaussian Elimination

Gaussian elimination provides the most efficient known method for solving sparse Boolean systems, where Boolean operations replace the real arithmetic used when solving linear systems [13,20]. Figure 1 shows a sketch of the Gaussian elimination algorithm. The code has two parts: forward elimination and back substitution. Forward elimination successively modifies the system structure, each time eliminating a vertex and all incident edges, and possibly adding edges between some remaining vertices. These structural modifications give Gaussian elimination its performance advantage over iterative methods. Eliminating a vertex v_i (termed the "pivot") involves updating the value of $b(v)$ for each uneliminated neighbor v of v_i. Then for each pair of uneliminated neighbors u and v, the value of $A(u,v)$ is updated. This may require adding a new "fill-in" edge to the graph if (u,v) does not already exist. During back substitution, the vertices are processed in the reverse of their elimination ordering. For each vertex v_i, the value of $x(v_i)$ is computed in terms of $b(v_i)$ and the value of $x(u)$ for every neighboring vertex u eliminated after v_i.

The code of Figure 1 also defines some notation to assist the proof of correctness and the performance analysis. To summarize, the vertices of V are labeled $v_1, \ldots, v_n$ in the order they are eliminated. The set $V_i \subseteq V$ is defined as the set of all vertices eliminated after v_i. The set E_i is defined as the set of all edges (both original and fill-in) connecting vertices in V_i. In the actual implementation, little of this information need be stored explicitly. Edges can be represented by adjacency lists with fill-in edges appended to the lists. A stack can keep track of the elimination ordering for use in back substitution. The set of uneliminated vertices can be represented by a priority data structure to implement the desired pivot selection rule.

Theorem 4 *The Gaussian elimination algorithm of Figure 1 solves the system* $[A,b]$.

A proof of this theorem is given in Appendix A. It involves showing that the elimination of a vertex does not change the solution for the remaining vertices. The final system involves only one vertex and is solved trivially. Each back substitution step then adds back a vertex, computing its solution in terms of those for the other vertices.

5.2 Pivot Selection

The efficiency of Gaussian elimination depends largely on the number of uneliminated neighbors each vertex has as it is eliminated. Consider a graph with n vertices. Assume for simplicity that the graph is ***structurally symmetric***, that is $(u,v) \in E$ whenever $(v,u) \in E$. This requirement can be met by adding edges to the graph with labels equal to 0.[2] With this simplification, an undirected graph describes the structure of the system to be solved. Referring to Figure 1, define the ***elimination degree*** of vertex v, denoted $d(v)$, as

$$d(v_i) = \left|\{u \in V_i | (u, v_i) \in E_{i-1}\}\right|.$$

[2]In fact such edges are often added to simplify the data structures, as it eliminates the need to store explicit pointers in the reverse direction of the edges.

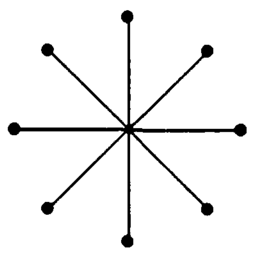


Figure 2: **Star Graph Example.** The elimination complexity of this class of graphs ranges between linear and cubic, depending on the elimination ordering.

The number of algebraic operations ($\wedge$ and $\vee$) for elimination is at most

$$2 \sum_{1 \le i \le n} [d(v_i) + d(v_i)^2].$$

This formula is highly sensitive to the values of $d(v)$. For example, if the degrees are all bounded by a constant, then only $O(n)$ operations are required. On the other hand, in the worst case $d(v_i)$ can equal $|V_i|$ for all vertices and $O(n^3)$ operations are required.

The vertex elimination degrees depend greatly on the elimination order. Consider, for example a "star" graph, such as the one shown in Figure 2. If the center vertex is eliminated first, fill-in edges are added between every pair of remaining remaining vertices, and the algorithm requires $O(n^3)$ operations. If, on the hand, this vertex is eliminated last, we would have $d(v) \le 1$ for all v, requiring only $O(n)$ operations.

Much has been written on strategies for choosing elimination orderings, including both empirical [21,22] and theoretical [23] results. In general, the problem of selecting an *optimal* ordering is NP-complete [18]. However, we would be satisfied with a "good", but not necessarily optimal, ordering, and we can exploit properties of the graphs that arise in MOS circuit analysis.

The following heuristic strategy is often cited [21,22] for deciding which vertex to select as the next pivot during Gaussian elimination:

Rule M: Choose a vertex that minimizes $d(v)$.

This rule is an example of a "greedy" strategy. That is, it selects a pivot to minimize the computational effort of the next step without regard to future eliminations. For MOS circuits, this strategy works quite well—with only a few exceptions the resulting elimination requires only $O(n)$ operations. A MOS circuit maps into a "channel graph" for symbolic analysis[7]. This graph contains a vertex for each storage (i.e., non-input) node n, and an edge (m, n) for each pair of storage nodes m, n forming the source and drain of a transistor. In general, this graph will contain many components, and each component is analyzed separately.

Most channel graphs describing digital MOS circuits fall into a restricted class that we shall term "general series-parallel" (GSP). This class extends conventional series-parallel

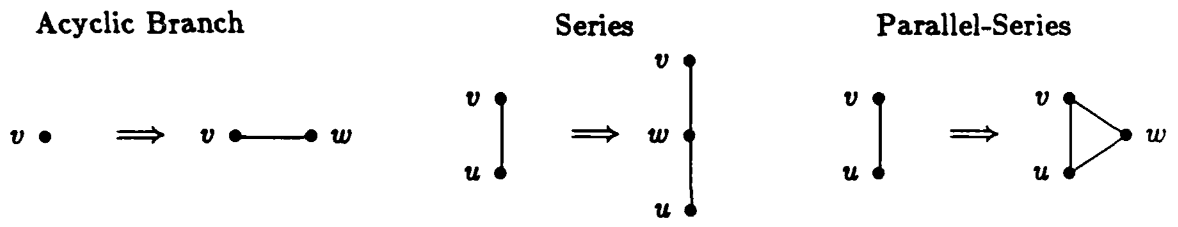


Figure 3: **General Series-Parallel Production Rules.** Any GSP graph can be generated by starting with a single vertex and applying a sequence of these rules.

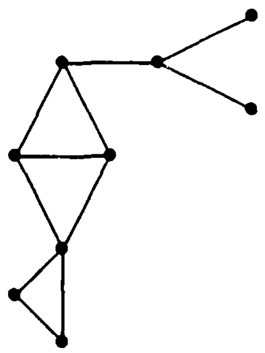


Figure 4: **Channel Graph for Complex nMOS Gate.** Even though the gate has a bridge in its pulldown network, the graph is general series-parallel.

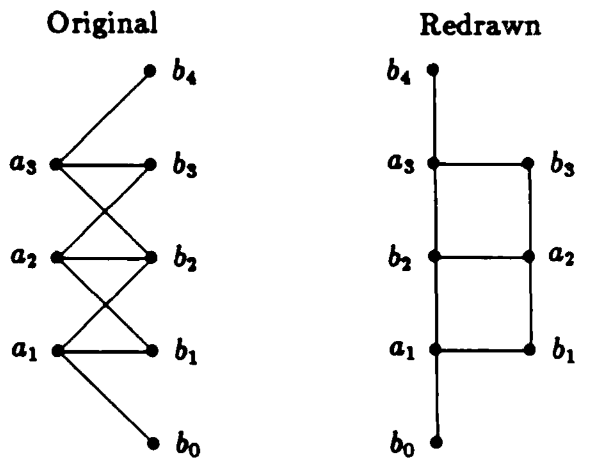


Figure 5: **Channel Graph for Shift Network.** Redrawing it shows the general series-parallel structure more clearly.

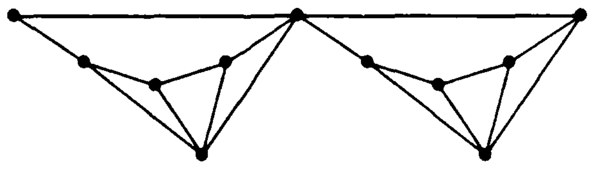


Figure 6: **Channel Graph for Section of Carry Chain.** Although not GSP, no vertex has elimination greater greater than 3.

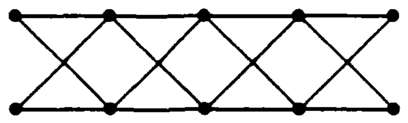


Figure 7: **Channel Graph for Parity Ladder.** Although not GSP, no vertex has elimination degree greater than 3.

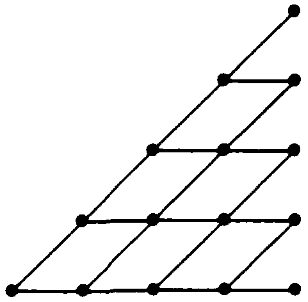


Figure 8: **Channel Graph for Tally Circuit.** This class of graphs has $O(n^2)$ elimination complexity when pivots are selected by Rule M.

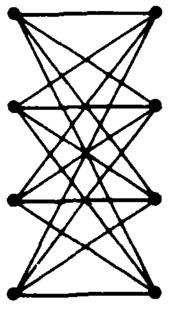


Figure 9: **Channel Graph for Barrel Shifter.** This class of graphs has $O(n^3)$ elimination complexity regardless of pivot sequence.

graphs[24] to include those containing acyclic branches. GSP graphs can be defined inductively starting with a single vertex as the basis, and applying the production rules illustrated in Figure 3. That is, given a GSP graph containing a vertex v, a new vertex w and edge (v, w) can be added to give a new GSP graph. Similarly, given a GSP graph containing vertices u and v and edge (u, v), a new vertex w along with edges (v, w) and (u, w) can be added. The edge (u, v) is either deleted for the Series production rule or retained for the Parallel-Series rule. We do not define a pure Parallel rule to avoid creating multigraphs. It can easily be seen that this class of graphs has significance for MOS circuits. Most MOS circuits involve transistors connected series-parallel to implement logic functions and acyclically to implement information transfer.

Figures 4 and 5 show examples of GSP channel graphs. Figure 4 is typical of those that arise when analyzing complex nMOS logic gates with connected pass transistors. Note that the graph contains no edge corresponding to the pullup transistor, since this transistor is connected directly to an input node. The pulldown network contains a "bridge", and hence many would not consider this a series-parallel graph. Most presentations of series-parallel networks assume a "virtual edge" between the top and bottom terminals (to represent the power supply.) Switch graphs need not include such an edge, and hence the channel graph is GSP. CMOS logic gates usually have GSP channel graphs as well. Figure 5 is typical of those that arise when analyzing pass transistor shift networks. This network transfers a set of inputs on the left to the outputs on the right shifted either -1, 0, or 1 positions. As normally drawn, the graph appears quite complex. However, it can be redrawn as shown on the right, making it easier to see the GSP structure. Experience has shown that many seemingly complex circuits have simple channel graphs.

Theorem 5 *A system of equations defined on a graph can be solved by Gaussian elimination such that no vertex has elimination degree greater than 2 if and only if the graph is general series-parallel.*

A proof of this theorem is given in Appendix A. It follows from the observation that the production rules of Figure 3 and the graph transformations caused by eliminating a vertex of degree less than or equal to 2 are inverses of each other. Our application requires only the "if" part of the theorem. The "only if" part is included for intellectual interest. It shows that only GSP graphs satisfy this bound on the maximum elimination degree.

Corollary 2 *Gaussian elimination applied to an n vertex GSP graph with pivots selected by Rule M requires at most $12n$ algebraic operations.*

An n vertex GSP graph must have between $n - 1$ and $2n - 1$ edges. Hence, this result shows that an the analysis of an s switch network requires $O(s)$ operations when the network has a GSP structure.

A survey of 4 books on VLSI [25,26,27,28], plus a direct analysis of many circuit designs has uncovered only a handful of non-GSP channel graphs, as illustrated in Figures 6, 7, 8 and 9. Figure 6 shows the graph for a section of the carry chain circuit from the MIPS-X processor [29]. Even when repeated for a number of stages, systems with this graph have linear elimination complexity, because no vertex has elimination degree greater than 3. The same holds for pass transistor parity ladders based on a well known relay contact network [1], as illustrated in Figure 7. In contrast, path enumeration over this graph gives a result of exponential complexity, while iterative methods have quadratic worst case complexity. Graphs that arise when a circuit is created by repeating a structure in one dimension generally have some constant upper bound on elimination degree and hence linear elimination complexity.

The Tally network of Mead and Conway [25], with graph illustrated in Figure 8 does not yield such favorable results. This network has the lower triangle of a square mesh as its channel graph. For such meshes, informal experiments indicate that selecting pivots by Rule M gives quadratic complexity. For a planar graph such as this, pivot selection by nested dissection can solve an s switch system with $O(s^{3/2})$ operations [23]. In practice, however, only small versions of this circuit are used, or restoring buffers are inserted for performance reasons. Either case yields small channel graphs, and Rule M handles small graphs well. For example, the four input tally circuit shown in the figure has maximum elimination degree 3.

A variety of pass transistor shift networks yield non-GSP channel graphs. A barrel shifter as shown in Figure 9 provides the most extreme case. The channel graph for this circuit is a complete bipartite graph. For solving such a dense system $O(n^3)$ operations are required for any elimination ordering. Given that the number of switches s grows quadratically with the number of nodes, however, the elimination complexity is a respectable $O(s^{3/2})$.

Other shift networks have complexity between those of Figures 5 and 9. For example, the Caltech Mosaic processor [30] has a network that passes the data either straight through, shifted 1 position, or shifted 4 positions, where shifts are circular. When following Rule M, experiments indicate that the elimination degree never exceeds 12 for such a graph, regardless of the shifter width. Although this bound yields a solution of linear complexity, the constant of proportionality becomes noticeably high. Fortunately, shift networks constitute only a small fraction of the total circuitry in a full scale VLSI chip. Even subnetworks with $O(n^3)$ elimination complexity should have little impact on the total result. Furthermore, this polynomial worst case complexity compares favorably to the exponential complexity of other methods.

As an aside, this analysis shows that Gaussian elimination would provide an efficient method for computing node voltages in a linear switch simulator such as RSIM [11]. On the other hand, the results do not carry over as well to circuit simulators such as SPICE [22]. When an implicit integration method is used in a circuit simulator, a conductance is inserted across the terminals of each capacitor. This effectively creates a connection between the gate, source, and drain of every MOS transistor. The resulting graphs can have far more complex structure than channel graphs.

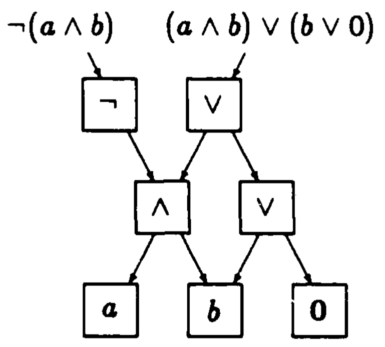


Figure 10: **DAG Representation of Two Formulas.** The leaves denote variables and constants, while the nodes denote Boolean operations. A formula is denoted by a pointer to a node.

6 Boolean Formula Representation and Manipulation

Up to this point, the presentation has intentionally remained vague as to how Boolean formulas are represented and manipulated. In fact, there are many possible representations offering a wide range of capabilities and limitations. As has been shown, most networks arising in MOS circuit analysis require a linear number of algebraic operations to analyze. Ideally, the Boolean formulas should be represented in such a way that the total size of the formulas for a network preserves this linear growth. A directed acyclic graph representation satisfies this requirement.

6.1 DAG Representation of Formulas

A directed acyclic graph (DAG) [31,32] resembles a parse tree, with leaves representing either variables or constants, and with internal nodes representing Boolean operations. In a DAG, however, a given subgraph may be shared by several branches, yielding a more compact representation. During the analysis of a switch network, the program constructs a single DAG having multiple roots. A formula is indicated by a pointer to some DAG node, where the formula denoted consists of the node and all of its descendants. Figure 10 shows a DAG representing two formulas.

During Gaussian Elimination, the program can perform an operation symbolically by simply adding a new node to the DAG with branches to the nodes representing the arguments. As observed by Tarjan [20], the DAG produced by this method can grow no larger than the total number of algebraic operations.

6.2 Formula Simplification

As the example of Figure 10 shows, formulas can often be simplified by applying laws of Boolean algebra. Fortunately, the DAG representation forms an ideal data structure for performing these simplifications and for detecting and eliminating common subexpressions [31]. In general, the problem of reducing a formula to its simplest form is NP-hard (proving tautology involves proving that a formula can be simplified to **1**). However, a large class of simplifications can be expressed in terms of local transformation on the DAG, where no transformation increases the number of nodes. This paper presents only transformations appropriate for the formulas generated by switch network analysis. In particular, it includes only a limited set of negation rules, because negation can only occur within the control formulas for the switches.

The program can readily apply simplifying transformations each time an operation is requested. That is, if some operation op (either $\wedge$, $\vee$, or $\neg$) is to be applied to a list of arguments A, the procedure applies transformations to produce a new list A', possibly changing the operation to op' as well. Then the procedure checks a symbol table (e.g., a hash table) to see if a node with this operation and with these arguments already exists. If not it creates a new node and stores a pointer to it in the symbol table. This method avoids ever creating duplicate subexpressions.

6.3 Simplification Rules

This presentation utilizes the following node representation. Associated with each node is a $type \in \{\wedge, \vee, \neg, \mathbf{0}, \mathbf{1}, var\}$. Types $\wedge$, $\vee$, and $\neg$ represent operations. Types **0** and **1** represent constants, while type var represents a variable. Associated with a node x for which $type(x) \in \{\wedge, \vee, \text{ or } \neg\}$ is a list of arguments $Args(x)$. Although the list is not technically a set (because it is ordered and contains duplicates), set notation is used to denote list operations. The set of all nodes is assumed totally ordered, as can be implemented by assigning a unique integer identifier to each node and ordering nodes by their identifiers. This total ordering serves only to permit a canonical listing of all children of a node.

The steps below outline a procedure to apply operation $\vee$ to a list of arguments A, where each argument is specified by a DAG node. The steps to apply $\wedge$ are similar, interchanging the roles of $\wedge$ and $\vee$, as well as those of **0** and **1**. Each step indicates an algebraic identity and an associated set of transformations. The steps are ordered in such a way that the procedure need only sequence through the list once to implement the operation.

1. Associativity: $(a \vee b) \vee c = a \vee (b \vee c)$
   For each $x \in A$ such that $type(x) = \vee$, remove x from A and add $Args(x)$ to A. This transformation guarantees that no node in the DAG will ever have a child of the same type. This transformation may or may not be desirable as is discussed below.

2. Commutativity: $a \vee b = b \vee a$
   Sort the elements of A. This transformation guarantees that all nodes in the DAG will list their children in the same order.

3. Idempotency: $a \vee a = a$
   Remove any duplicate entries from A. Since the elements of A are sorted, duplicates must appear consecutively.

4. Identity: $a \vee \mathbf{0} = a$
   Remove from A any element x such that $type(x) = \mathbf{0}$.

5. Annihilator: $a \vee \mathbf{1} = \mathbf{1}$
   If A contains any element x such that $type(x) = \mathbf{1}$, then return x as the result of the evaluation.

6. Excluded Middle: $a \vee \neg a = \mathbf{1}$
   If A contains elements x and y such that $type(x) = \neg$ and $y \in Args(x)$, then return a node of type $\mathbf{1}$ as the result of the evaluation.

7. Redundancy: $b \leq a \;\Rightarrow\; b \vee a = a$.
   For each $x \in A$, label x with 1 and every $y \in A - \{x\}$ with 0. If a search with these labels leads to a contradiction, then remove x from A. The search procedure is described in detail below.

8. Degenerate Cases: $\bigvee_{a \in \{b\}} a = b$, $\bigvee_{a \in \emptyset} a = \mathbf{0}$

   (a) If A contains only a single element x, then return x as the result of the evaluation.

   (b) If A is empty, then return a node of type $\mathbf{0}$ as the result of the evaluation.

9. Common Subexpressions
   Look in the symbol table for an entry with key $\langle \vee, A \rangle$.

   (a) If an entry is found, then return it as the result of the evaluation.

   (b) If no entry is found, then create a new node x with $type(x) = \vee$ and $Args(x) = A$. Add an entry for x to the symbol table with key $\langle \vee, A \rangle$. Return x.

   By this method, we guarantee that duplicate nodes are never created.

6.4 Discussion of Transformations

Observe that this list of transformations does not include any for the two distributive laws:

$$\begin{aligned}(a \vee b) \wedge c &= (a \vee c) \wedge (b \vee c)\\ (a \wedge b) \vee c &= (a \wedge c) \vee (b \wedge c).\end{aligned}$$

If we were to apply transformations that distribute one operation over the other, the size of the DAG would be increased. The DAG could even grow to exponential size, if for example, distributivity were applied to transform the formula into sum-of-products form. On the other hand, we could attempt to recognize opportunities to factor expressions.

However, expressions such as $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$ can be factored in more than one way, giving different degrees of simplification. Thus, the manipulator ignores the distributive laws altogether.

The associativity transformation (step 1) does not increase the number of nodes in the DAG, and hence incurs no added complexity under a *node cost* model, where the complexity of a DAG is expressed as the total number of nodes. However, it can create nodes with more children than the number of arguments in the original list. For example, the evaluation sequence

$$\begin{array}{rcl} x & \leftarrow & a \wedge b \\ y & \leftarrow & x \wedge c \\ z & \leftarrow & y \wedge d \end{array}$$

would create 3 nodes having 2, 3, and 4 children, respectively. The *binary cost* model expresses the DAG complexity as the sum of its node costs, where a node with n children has cost $n - 1$. This measure corresponds to the number of binary operations required to evaluate the resulting expression. Under this cost model, the associativity transformation can increase the complexity. The above example would yield a DAG of binary cost 6, whereas omitting the transformation would yield a DAG of cost 3.

The associativity transformation also interacts with the redundancy transformation (step 7), described in detail later. This transformation requires a search of the DAG for each element of the list A, incurring a significant computational effort. The associativity transformation can expand this list and, consequently, the number of searches.

Of course, omitting this transformation causes the manipulator to overlook some useful simplifications. It will fail to recognize the equivalence of some expressions. Furthermore, it will fail to eliminate some redundancies that would otherwise be found. For example, consider the DAG for the expression $a \vee [c \vee (a \wedge b)]$. The associativity transformation would create a list of arguments a, c, and $a \wedge b$. The redundancy transformation would then eliminate the third argument, yielding a simplified expression $a \vee c$. On the other hand, no simplification would occur if the associativity transformation were omitted, because neither the expression a nor $c \vee (a \wedge b)$ is redundant with respect to the other.

The choice of whether or not to apply the associativity transformation depends on the nature of the formulas generated and the appropriate complexity measure for the DAG. Our experiments with a symbolic MOS circuit analyzer clearly indicate that, under the binary cost model, the associativity transformation increases the DAG complexity by a factor of at least 2 for almost all circuits. Furthermore, depending on the particular circuit, it can greatly increase the amount of CPU time spent searching for redundancies. However, omitting the transformation yields formulas with a noticeable number of redundant terms. Hence the desirability of the transformation depends on the intended application of the symbolic analyzer output.

6.5 Redundancy Testing

The redundancy test mentioned in step 7 has proved important when simplifying the formulas arising during MOS circuit analysis. Due to the way the program analyzes a

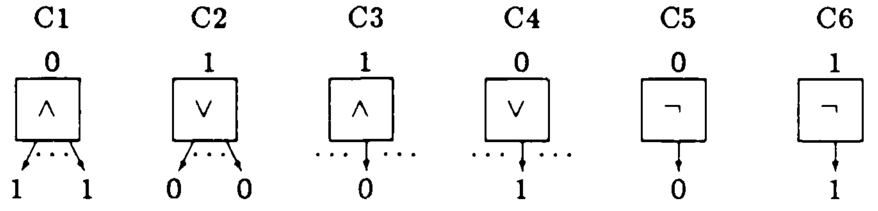


Figure 11: **Contradiction Rules for Redundancy Test**. The search terminates successfully when one of these labelings arises.

MOS network by solving a series of switch networks, it would otherwise construct complex formulas containing many redundancies. Methods for discovering redundancy range widely in their completeness and efficiency. On one extreme, a method that reliably detects any case where 2 formulas are ordered $x \leq y$ can solve the NP-hard equivalence problem. That is, two formulas are equivalent if and only if both $x \leq y$ and $y \leq x$. On the other extreme, simple graph transformations can apply the simple absorption rule $a \vee (a \wedge b) = a$. Simple approaches, however, miss many opportunities for simplification.

The method discussed below strikes a compromise between efficiency and completeness. It applies a search technique that attempts to prove that an argument is redundant but applies tight controls to avoid combinatorial complexity.

Proposition 4 states that two Boolean formulas are ordered $x \leq y$ if and only if no assignment of 1's and 0's to the variables can cause x to evaluate to 1, while y evaluates to 0. The redundancy test attempts to prove this property by contradiction, in a manner reminiscent of an automatic theorem prover based on the Resolution Principle [33]. That is, it assigns value 1 to x, 0 to y, and determines the logical consequences of these assignments. If it reaches a contradiction, then the formulas are ordered, otherwise they are assumed unordered. The manipulator applies this test to each argument x in the list A in an attempt to drop the argument from the list. That is, for an $\vee$ (respectively, $\wedge$) operation, the test searches for a contradiction with x assigned value 1 (resp., 0) and all other arguments assigned 0 (resp., 1).

The redundancy test requires augmenting the DAG data structure with an additional *value* field for each node, set to either 0, 1, or X (indicating an unknown value). Initially, the nodes in argument list A are set to 0 or 1 specifying the proof goal, while the other nodes are set to X. Each node also has a list of pointers to its parents in the DAG. The program searches for a contradiction in a manner similar to the implication step of the D-algorithm used in test generation [34]. A queue, initialized with the nodes in argument list A plus their parents, holds nodes that are candidates for further inferences. Boolean values are propagated through the DAG by repeatedly removing a vertex from the queue and applying inference rules that may change the node value or that of one or more child. If a node value changes the program adds either its children or parents to the queue as candidates for further inferences. This process continues until either a contradictory

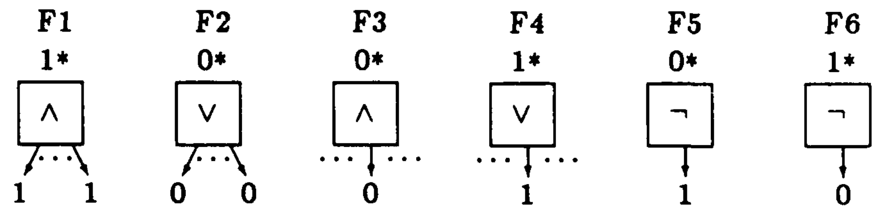


Figure 12: **Forward Inference Rules for Redundancy Test.** These rules change the value of a node based on those of its children. Asterisks mark the values changed from X to 0 or 1.

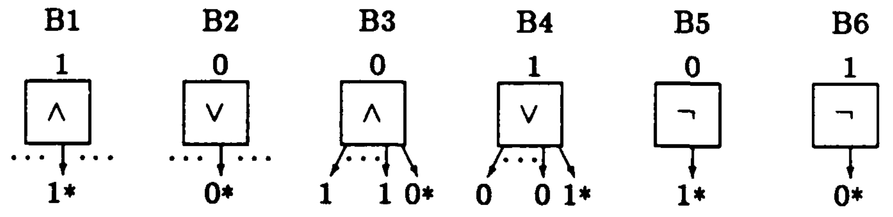


Figure 13: **Backward Inference Rules for Redundancy Test.** These rules change the value of a child based on those of its parent and siblings. Asterisks mark the values changed from X to 0 or 1.

labeling is encountered (success), or the queue becomes empty (failure).

Figure 11 illustrates the set of contradictory labelings that cause the search to terminate successfully. In this figure, the label above the node indicates the value associated with the node, while the labels below indicate the values associated with the children. Note that rules C1 and C2 require all children to have a particular value, while rules C3 and C4 require only a single child with the specified value.

Figures 12 and 13 present the set of inference rules by which Boolean values are propagated through the DAG. For each rule, an asterisk indicates the value changed by the rule from X to 0 or 1. Figure 12 illustrates the set of *forward* inference rules, i.e., those that cause the value of a node to change based on the values of its children. For example, rule F1 indicates that if all children of an $\wedge$ node have value 1, then it too must have value 1. Rule F3 indicates that if an $\wedge$ node has any child with value 0, then it must have value 0. Successful application of a forward inference rule to a node causes queueing of any parent not already in the queue.

Figure 13 illustrates the set of *backward* inference rules, i.e., those that cause the value of one or more child to change based on the value of the node and possibly the values of the other children. For example, rule B1 indicates that if an $\wedge$ node has value 1, then every child must have value 1. Rule B3 indicates that if an $\wedge$ node has value 0 and all but one child have value 1, the remaining child must have value 0. Successful application of a backward inference rule to a node causes queuing of any child whose value changes and is not already on the queue. Any other parent of a child whose value changes is also queued, unless it is already in the queue.

Figure 14 shows an example of how the redundancy test can prove that two formulas x and y are ordered $x \leq y$. This example was adapted from one that can actually occur during the symbolic analysis of a MOS circuit, demonstrating the need for a sophisticated redundancy test. The figure does not show the descendants of the nodes labeled with operation *op*, as they are not required to prove redundancy. This example arises when a back substitution step of Gaussian elimination requires an evaluation of the form $y \leftarrow x \vee y$. As a result of the successful redundancy test, formula y remains unchanged, yielding a significant simplification. For purpose of discussion, each node is labeled with an identifying letter to its left. The initial values assigned to the nodes are shown to their right. The queue initially contains nodes **a**, **d**, and **c**. The search proceeds by the series of steps shown in Table 1. It terminates once a conflicting labeling is found at node **e**.

The following analysis of the search algorithm shows that it has linear complexity, as measured in the total number of branches in the DAG. The search only queues a node when the value on the node, one of its parents, or one of its children changes from X to 0 or 1. Each branch in the DAG can cause the nodes on either end to be queued at most once. Therefore, the total number of queuing operations cannot exceed twice the number of branches in the DAG. The set of inference rules can be applied to a node in constant time, if counts are maintained for each node specifying the number of children with value 0, 1, and X. Hence the algorithm has time complexity linear in the DAG size. Furthermore, the constant of proportionality is quite small.

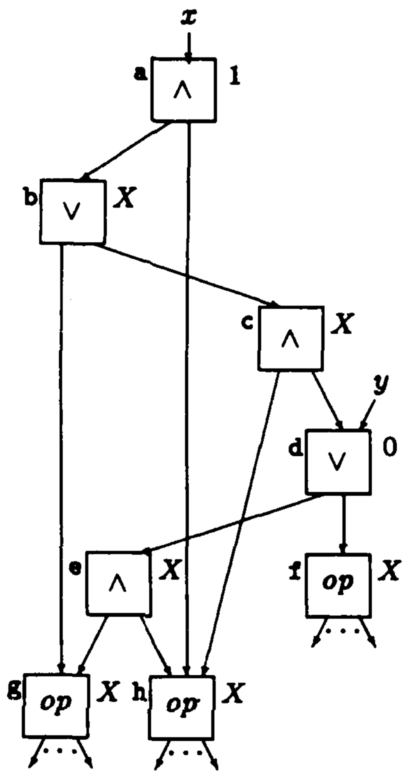


Figure 14: **Redundancy Search Example.** The search proves that formula x is redundant with respect to formula y. Nodes are labeled by their values at the start of search and by letters for discussion in the text.

Step	Node	Rule	Changed Nodes	New Value	Queued Nodes
1.	a	B1	b, h	1	b, h
2.	d	B2	e, f	0	e, f
3.	c	F3	c	0	
4.	b	B4	g	1	g
5.	h	none			
6.	e	C1			

Table 1: **Inference Sequence for Redundancy Search Example.** The search finds a contradiction at node e.

However, since the search must be initiated once for each argument every time a Boolean operation is performed, the total time spent searching for redundancies can become quite large. Our implementation controls the time spent searching in 2 ways. First, the search need only consider nodes that are descendants of the nodes in the argument

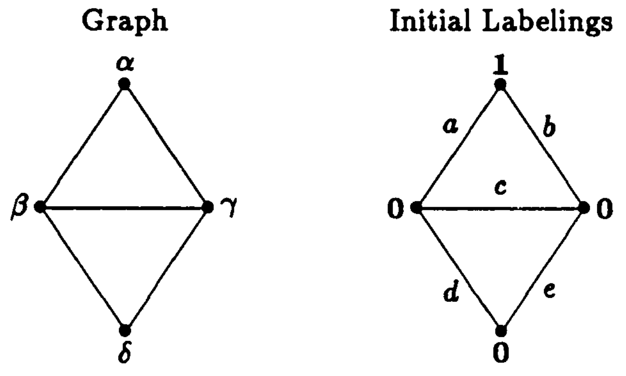


Figure 15: **Example System of Equations.**

list A. The program avoids visiting extraneous nodes by keeping the value fields of the nodes initialized to a special value indicating "unreachable." Before a search begins, the program traces all descendants of the nodes in A and sets their values to X. The search then only visits nodes with values X, 0, or 1, and upon termination resets all nodes to "unreachable". This constraint will not reduce the success rate of the search. Second, the search proceeds in breadth-first order (by using a first-in, first-out queueing discipline), and can be constrained to give up once it reaches a specified depth. This constraint reduces the success rate of the search, but eliminates cases requiring extensive search having little chance of success. With appropriate constraint, experience indicates that this approach to redundancy testing yields significant simplifications for a reasonable computational effort.

It must be emphasized, however, that the redundancy test is not complete. For example, it will recognize that $a \wedge (b \vee d)$ is redundant with respect to $(a \wedge b) \vee (a \wedge d)$, but not *vice-versa*, even though the two expressions are equivalent. The algorithm could be extended to one that detects all redundant cases by adding combinatorial search and backtracking. However, this could greatly increase the computational effort, especially considering that in most cases the redundancy test will fail.

7 Symbolic Analysis Example

As with many algorithms designed for computer implementation, this analysis method is very tedious to execute by hand. For systems of significant size, the DAG becomes far to large to draw. Small systems, on the other hand, lend well to exhaustive path analysis. Hence it is hard to demonstrate the advantages of our method with an example. With these limitations in mind, several useful insights can be gained from a simple example.

Figure 15 shows the graph corresponding to a bridge network with source terminal α. In this example, the edge labeling is symmetric: $A(u,v) = A(v,u)$, and hence the system structure can be represented by an undirected graph. The steps of Gaussian elimination

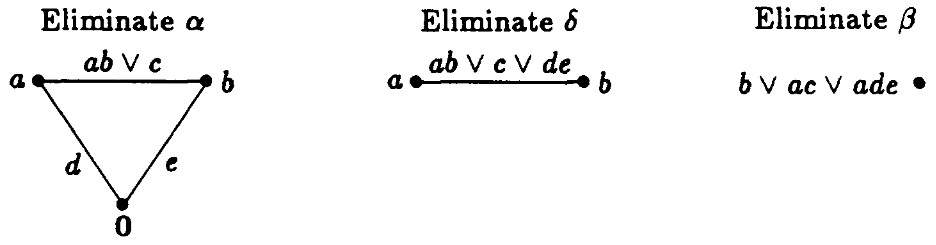


Figure 16: **Elimination Steps for Example System.**

preserve this symmetry, and a straightforward modification of the elimination code reduces the number of operations by almost a factor of 2. Figure 16 shows the sequence of labelings produced by the successive elimination steps. For readability, the Boolean formulas were simplified by hand and are shown without $\wedge$ symbols. The back substitution steps yield:

$$\begin{aligned} x(\gamma) &= b \vee ac \vee ade \\ x(\beta) &= a \vee bc \vee bde \\ x(\delta) &= ad \vee bcd \vee be \vee ace \\ x(\alpha) &= 1 \end{aligned}$$

Figure 17 shows the complete DAG produced in analyzing this system. For readability, the branches to nodes representing variables are indicated by the variable names. This DAG looks very complex, and it is difficult to verify that it correctly characterizes the network. Observe, however, that this representation of the formulas involves only 10 Boolean operations. The formulas derived by hand simplification appear much more readable, but they require a total of 11 Boolean operations. Furthermore, under the binary cost model, (a better measure of the evaluation cost for a set of formulas), the DAG has cost 11, whereas a straightforward implementation of the hand-derived formulas has cost 19. Only with considerable effort can the hand-derived formulas be transformed into ones involving a total of 11 binary operations. This example shows that our method produces results that are very compact although difficult for humans to read. Compactness counts more for results that are used by other computer programs.

8 Conclusion

This paper has shown that a careful choice of algorithm and data structures yields a far more efficient solution than does a naive approach. Furthermore, by casting the problem in terms of systems of equations, the wealth of knowledge that has accumulated about solving linear systems could be applied.

The symbolic analysis technique described in this paper has a wide range of applications beyond switch networks. Direct methods such as Gaussian elimination are examples of

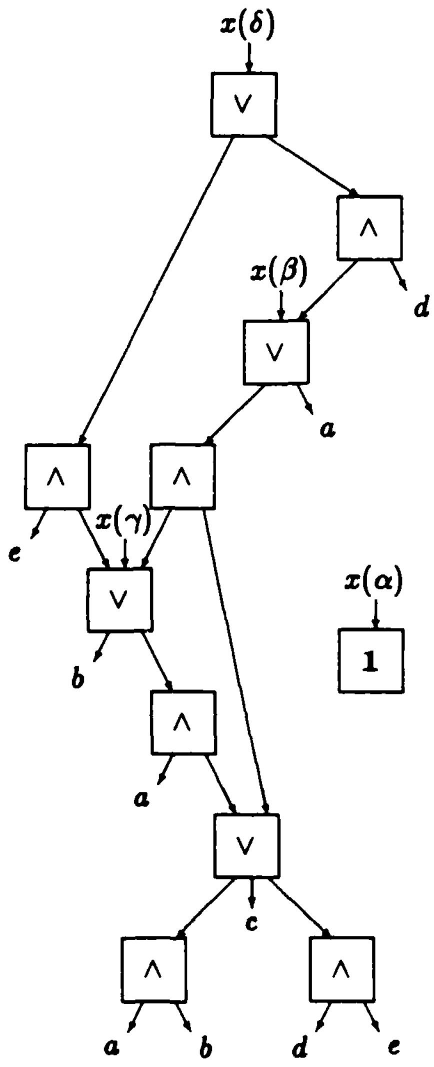


Figure 17: **DAG Produced in Analyzing Example Network.** Although appearing more complex than the formulas derived by hand, the DAG representation is actually more compact.

oblivious algorithms. That is, the control sequence depends only on the graph structure of the system to be solved and not on the data values.[3] In contrast, iterative methods perform data-dependent branches when testing for convergence or when deciding which vertex to update next. Any oblivious algorithm can be executed symbolically to yield some explicit representation (e.g., a DAG) of the output generated by the program for all possible

[3]Although data-dependent pivoting may be required when solving linear systems for numerical reasons.

input data. Such a preprocessing step can yield a significant performance advantage in applications that must evaluate many systems sharing a common structure but differing in data values. Furthermore, the representation generated by symbolic analysis can be executed by hardware that supports only the operations of the underlying algebra, rather than general purpose computation. Hardware that supports only restricted domains such as Boolean operations can achieve very high performance at a reasonable cost. Problems that can be solved by Gaussian elimination and hence are amenable to symbolic analysis include: linear systems, shortest path calculations, bottleneck flow path calculations, and conversion of finite automata to regular expressions [16].

Solving path problems by Gaussian elimination becomes especially attractive as computers with parallel processing capabilities become available. The "greedy" pivot selection rule presented here gives very good results in terms of the total size of the formulas. If the results are to be executed on machines that support high degrees of parallelism, however, the potential concurrency allowed by the formulas should be maximized as well. As the DAG of Figure 17 illustrates, greedy pivot selection tends to yield "long, skinny" formulas without much potential for concurrent evaluation. On the other hand, pivot selection based on nested dissection [23,35] yields "short, fat" formulas, many terms of which could be evaluated simultaneously. In particular, the family of GSP graphs satisfies a *2-separator* theorem, meaning that it is always possible to find 2 vertices whose removal would split the graph into two GSP graphs of roughly equal size. For such graphs, nested dissection yields formulas with $O(n)$ operations, although the constant of proportionality would be somewhat higher. However, the formulas also have maximum depth $O(\log n)$, giving sublinear evaluation times if sufficient resources are available. Various other graph classes lead to formulas with sublinear maximum depth. In contrast, no iterative method can give sublinear performance for the graph structures of interest regardless of the processing hardware.

Symbolic analysis, as presented here, simply transforms one description of a Boolean computation into another, that is from a switch network to a set of formulas. Some applications, such as proving two networks equivalent or that a network implements a given function, require stronger results. These problems are NP-hard [18], and many believe efficient algorithms for such tasks do not exist. However, several approaches yield practical results in many instances. One approach uses a different representation of Boolean functions that makes equivalence testing more straightforward. The author [36] has devised a representation based on a different type of directed acyclic graph that is canonical, i.e., a given function has a unique representation. Equivalence testing then becomes a simple matter of testing whether two graphs match exactly. Furthermore, many of the functions encountered in logic design applications are represented by reasonably small graphs. Symbolic analysis could also be performed using these graphs as the underlying data structure for representing Boolean functions, yielding a canonical description of the network function.

A Proofs of Theorems

A.1 Systems of Boolean Equations

Theorem 1 *Any Boolean system $[A, b]$ has a unique solution x given by the limit to the sequence x^i, where $x^0(v) = \mathbf{0}$ for all v, and $x^i(v)$ is defined for all $i > 0$ and all v as:*

$$x^i(v) \;=\; b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]. \tag{4}$$

Proof: First, we will show by induction on i that $x^i \leq x^{i+1}$. The basis clearly holds, because $x^0(v) = \mathbf{0} \leq x^1(v)$. Assuming by induction that $x^i(u) = x^i(u) \vee x^{i-1}(u)$ for any vertex u, expanding Equation 4 for x^{i+1} and separating terms gives

$$\begin{aligned} x^{i+1}(v) &= b(v) \vee \bigvee_{(u,v)\in E} \Big([x^i(u) \vee x^{i-1}(u)] \wedge A(u,v)\Big) \\ &= \left[b(v) \vee \bigvee_{(u,v)\in E} [x^i(u) \wedge A(u,v)]\right] \vee \left[b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]\right] \\ &= x^{i+1}(v) \vee x^i(v). \end{aligned}$$

Thus, the sequence is nondecreasing. Since the domain $\mathcal{B}$ is finite, there must be some value k such that $x^k = x^{k+1}$.[4] From Equation 4 and by induction on i, it is easy to see that $x^k = x^i$ for all $i \geq k$, and consequently the sequence converges to a unique value. Furthermore, this vertex labeling clearly satisfies the system $[A, b]$.

Now suppose that some other labeling y satisfies the system $[A, b]$. We will show by induction on i that $y \geq x^i$ for all i. Clearly $y(v) \geq \mathbf{0} = x^0(v)$, and hence the basis condition holds. Now suppose that $x^{i-1}(u) \vee y(u) = y(u)$ for every vertex u. Then

$$\begin{aligned} y(v) &= b(v) \vee \bigvee_{(u,v)\in E} \Big([x^{i-1}(u) \vee y(u)] \wedge A(u,v)\Big) \\ &= \left[b(v) \vee \bigvee_{(u,v)\in E} [x^{i-1}(u) \wedge A(u,v)]\right] \vee \left[b(v) \vee \bigvee_{(u,v)\in E} [y(u) \wedge A(u,v)]\right] \\ &= x^i(v) \vee y(v) \end{aligned}$$

for every vertex v, indicating that $y \geq x^i$. Hence any labeling that satisfies $[A, b]$ must be greater than or equal to the limit of the sequence.

□

[4] In fact, k must be less than n.

Since this result holds for all paths to v, Proposition 2 shows that it must hold for their sum. Hence x is the minimum vertex labeling satisfying $[A, b]$.

□

Theorem 3 *x is the solution of the system $[A, b]$ if and only if $\overline{x}$ is the solution of the dual system $[\overline{A}, \overline{b}]^D$.*
Proof: DeMorgan's Laws can be generalized to the following rule for complementing a Boolean formula:

> Complement every variable, replace every $\wedge$ by $\vee$, every $\vee$ by $\wedge$, every $\mathbf{0}$ by $\mathbf{1}$, and every $\mathbf{1}$ by $\mathbf{0}$.

From this rule, we can see that the conditions for a labeling y to satisfy system $[A, b]$ are identical to those for $\overline{y}$ to satisfy the the dual system $[\overline{A}, \overline{b}]^D$. Furthermore, if $x \leq y$ for all y in some set Y, then $\overline{x} \geq \overline{y}$ for all $y \in Y$, and *vice-versa*. Therefore, the minimum labeling satisfying $[A, b]$ must equal the complement of the maximum labeling satisfying $[\overline{A}, \overline{b}]^D$, and *vice-versa*.

□

A.2 Gaussian Elimination

Theorem 4 *The Gaussian elimination algorithm of Figure 1 solves the system $[A, b]$.*
Proof: The key idea of the proof is to show that each time a vertex v_i is eliminated, a modified system $[A_i, b_i]$ is created over (V_i, E_i) such that the solution of this system equals the solution of the original for all $v \in V_i$. Assume for simplicity that there are no edges of the form (v, v) in E. It can easily be shown that removing such edges will not alter the system solution. Furthermore, the elimination code does not add any such edges as fill-in. We also adopt the convention that $A(u, v) = \mathbf{0}$ whenever $(u, v) \notin E$, and similarly that $A_i(u, v) = \mathbf{0}$ whenever $(u, v) \notin E_i$. Under these two conventions, Equation 1 can be written in two ways:

$$\begin{aligned} x(v) &= b(v) \vee \bigvee_{u \in V} [x(u) \wedge A(u, v)] \\ &= b(v) \vee \bigvee_{u \in V - \{v\}} [x(u) \wedge A(u, v)]. \end{aligned} \tag{6}$$

Define the system $[A_0, b_0]$ over the graph (V_0, E_0) as $A_0 = A$ and $b_0 = b$. For $n \geq i \geq 1$ define the system $[A_i, b_i]$ over the graph (V_i, E_i) as

$$A_i(u, v) = A_{i-1}(u, v) \vee [A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)] \tag{7}$$

and

$$b_i(v) = b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)]. \tag{8}$$

Observe that the definition of A_i preserves the property that $A_i(u, v) = \mathbf{0}$ when $(u, v) \notin E_i$, because E_i is guaranteed to have an edge (u, v) if both (u, v_i) and (v_i, v) are in E_{i-1}.

For $1 \leq i \leq n$, define the labeling x_{i-1} over the vertices of V_{i-1} as:

$$x_{i-1}(v) = \begin{cases} b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v_i)], & v = v_i \\ x_i(v), & v \in V_i \end{cases} \tag{9}$$

Note that $V_n = \emptyset$, and hence x_{n-1} is well defined. It can also be seen that the labeling x produced by the code of Figure 1 equals the labeling x_0 defined by Equation 9 for $i = 1$.

We will show by induction on i (starting from $n - 1$ and working downward) that x_i is the solution of the system $[A_i, b_i]$. Clearly x_{n-1} is the solution of $[A_{n-1}, b_{n-1}]$, because Equations 6 and 9 both reduce to $x_{n-1}(v_n) = b_{n-1}(v_n)$. Assume that x_i is the solution of the system $[A_i, b_i]$. We must show that x_{i-1} satisfies the system $[A_{i-1}, b_{i-1}]$, and that $x_{i-1} \leq y$ for any other labeling y satisfying this system.

For $v \in V_i$, given that x_i satisfies $[A_i, b_i]$, $x_i(v)$ can be expanded using Equation 6 as

$$x_i(v) = b_i(v) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_i(u, v)].$$

The definitions for $b_i(v)$ and $A_i(u, v)$ can then be substituted to give

$$\begin{aligned} x_i(v) = {} & b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)] \vee \\ & \bigvee_{u \in V_i} \Big[x_i(u) \wedge \Big(A_{i-1}(u, v) \vee [A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)]\Big)\Big]. \end{aligned}$$

Rearranging terms gives

$$\begin{aligned} x_i(v) = {} & b_{i-1}(v) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v)] \vee \\ & \left[\left(b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_i(u) \wedge A_{i-1}(u, v_i)]\right) \wedge A_{i-1}(v_i, v)\right] \end{aligned}$$

Substituting the definition for x_{i-1} gives

$$\begin{aligned} x_{i-1}(v) = x_i(v) &= b_{i-1}(v) \vee \bigvee_{u \in V_i} [x_{i-1}(u) \wedge A_{i-1}(u, v)] \vee [x_{i-1}(v_i) \wedge A_{i-1}(v_i, v)] \\ &= b_{i-1}(v) \vee \bigvee_{u \in V_{i-1}} [x_{i-1}(u) \wedge A_{i-1}(u, v)] \end{aligned}$$

For $v = v_i$, we can substitute $x_{i-1}(u)$ for $x_i(u)$ in Equation 9 giving

$$x_{i-1}(v_i) = b_{i-1}(v_i) \vee \bigvee_{u \in V_i} [x_{i-1}(u) \wedge A_{i-1}(u, v_i)].$$

Combining these two cases we see that x_{i-1} satisfies $[A_{i-1}, b_{i-1}]$.

Now suppose that a vertex labeling y defined over V_{i-1} satisfies the system $[A_{i-1}, b_{i-1}]$. Define the labeling y' over V_i as $y'(v) = y(v)$ for all $v \in V_i$. We will first show that

y' satisfies the system $[A_i, b_i]$, and therefore by the induction assumption that $x_{i-1}(v) = x_i(v) \leq y'(v) = y(v)$ for all $v \in V_i$. Then we will show that $x_{i-1}(v_i) \leq y(v_i)$, thereby completing the proof that $x_{i-1} \leq y$. For $v \neq v_i$, expanding $y(v)$ using Equation 6 and substituting the definition of y' gives

$$y'(v) \;=\; y(v) \;=\; b_{i-1}(v) \vee [y(v_i) \wedge A_{i-1}(v_i, v)] \vee \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v)].$$

Expanding $y(v_i)$ using Equation 6 gives

$$\begin{aligned} y'(v) \;=\;& b_{i-1}(v) \vee [b_{i-1}(v_i) \wedge A_{i-1}(v_i, v)] \vee \\ & \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v_i) \wedge A_{i-1}(v_i, v)] \;\vee\; \bigvee_{u \in V_i} [y'(u) \wedge A_{i-1}(u, v)]. \end{aligned}$$

Combining terms and substituting the definitions of b_i and A_i gives

$$y'(v) \;=\; b_i(v) \vee \bigvee_{u \in V_i} [y'(u) \wedge A_i(u, v)].$$

Therefore y' satisfies the system $[A_i, b_i]$. For $v = v_i$, we can assume that $x_{i-1}(u) \vee y(u) = y(u)$ whenever $A_{i-1}(u, v_i) \neq \mathbf{0}$. Hence, $y(v_i)$ can be expanded by Equation 6 as

$$\begin{aligned} y(v_i) \;&=\; b_{i-1}(v_i) \vee \bigvee_{u \in V_{i-1}} \Big([x_{i-1}(u) \vee y(u)] \wedge A_{i-1}(u, v_i)\Big) \\ &=\; \left[b_{i-1}(v_i) \vee \bigvee_{u \in V_{i-1}} [x_{i-1}(u) \wedge A_{i-1}(u, v_i)]\right] \vee \\ &\quad\; \left[b_{i-1}(v_i) \vee \bigvee_{u \in V_{i-1}} [y(u) \wedge A_{i-1}(u, v_i)]\right] \\ &=\; x_{i-1}(v_i) \vee y(v_i), \end{aligned}$$

and hence $x_{i-1}(v_i) \leq y(v_i)$. Thus we have shown that $x_{i-1} \leq y$ for any y satisfying the system $[A_{i-1}, b_{i-1}]$, completing the inductive proof that x_i is the solution of the system $[A_i, b_i]$. We have therefore proved the correctness of the algorithm, because the systems $[A, b]$ and $[A_0, b_0]$ are identical, and the labeling x returned by the algorithm equals x_0.

□

Theorem 5 *A system of equations defined on a graph can be solved by Gaussian elimination such that no vertex has elimination degree greater than 2 if and only if the graph is general series-parallel.*

Proof: Assume the graph (V, E) is constructed by a sequence of productions obeying the rules of Figure 3. Suppose the final step involves adding vertex w and one or more edges to the graph (V', E') and possibly deleting an edge. Then vertex w has degree less than or equal to 2 in (V, E). Furthermore, if w is selected as a pivot in Gaussian elimination, the resulting elimination operations will yield the graph (V', E'). This graph is also GSP. and hence the process can be continued until all vertices are eliminated.

Conversely, suppose Gaussian elimination can be performed for a system defined on the graph (V, E) such that no vertex has elimination degree greater than 2. Then with the graph (V_{n-1}, E_{n-1}) as the basis, where $V_{n-1} = \{v_n\}$ and $E_{n-1} = \emptyset$, we can construct the graph (V, E) by a sequence of production rules, adding vertices in the reverse of their elimination order. If vertex v_i has a single neighbor v in (V_{i-1}, E_{i-1}), then graph (V_{i-1}, E_{i-1}) is constructed from (V_i, E_i) by applying the Acyclic Branch rule with $w = v_i$. If vertex v_i has two neighbors u and v in (V_{i-1}, E_{i-1}), then graph (V_{i-1}, E_{i-1}) is constructed from (V_i, E_i) by applying either the Series or the Parallel-Series rule with $w = v_i$, depending on whether or not $(u, v) \in E_{i-1}$. This process proceeds until i reaches $(V_0, E_0) = (V, E)$.

$\square$

References

[1] C. E. Shannon, "A Symbolic Analysis of Relay and Switching Circuits", *Trans. of the AIEE*, Vol. 57 (1938), pp. 713–723.

[2] J. A. Brzozowski and M. Yoeli, *Digital Networks*, Prentice-Hall, 1976.

[3] M. Lightner and G. Hachtel, "Implication Algorithms for Switch Level Functional Macromodeling, Implementation, and Testing." *19th Design Automation Conf.*, IEEE (July, 1982), pp. 691–698.

[4] M. Yoeli, and J. A. Brzozowski, "A Mathematical Model of Digital CMOS Networks", *Canadian Conf. on VLSI* (1985).

[5] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers* Vol. C-33, No. 2 (February, 1984) pp. 160–177.

[6] J. Hayes, "A Unified Switching Theory with Applications to VLSI Design", *Proc. IEEE*, Vol. 70, No. 10 (October, 1982), pp. 1140–1151.

[7] R. E. Bryant, *Boolean Analysis of MOS Circuits*, companion paper (1987).

[8] F. E. Hohn and L. R. Schissler, "Boolean Matrices and Combinational Circuit Design", *Bell Systems Technical Journal*, Vol. 34 (1955), pp. 177–202.

[9] G. Ditlow, W. Donath, and A. Ruehli, "Logic Equations for MOSFET Circuits", *International Symposium on Circuits and Systems*, IEEE (1983), pp. 752–755.

[10] I. N. Hajj, and D. Saab, "Symbolic Logic Simulation of MOS Circuits", *International Symposium on Circuits and Systems*, IEEE (1983).

[11] C. J. Terman, *Simulation Tools for Digital LSI Design*, PhD Thesis, MIT Dept. Elec. Eng. and Comp. Sci. (October, 1983).

[12] E. Cerny, and J. Gecsei, "Simulation of MOS Circuits by Decision Diagrams", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-4, No. 4 (October, 1985), pp. 685–693.

[13] B. A. Carré, "An Algebra for Network Routing Problems", *J. Inst. Maths Applics.*, Vol. 7 (1971), pp. 273–294.

[14] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[15] D. J. Lehmann, "Algebraic Structures for Transitive Closure", *Theoretical Computer Science*, Vol. 4 (1977), pp. 59–76.

[16] R. E. Tarjan, "A Unified Approach to Path Problems", *J. ACM*, Vol. 23, No. 3 (July, 1981), pp. 577–593.

[17] M. A. Harrison, *Introduction to Switching and Automata Theory*, McGraw-Hill, 1965.

[18] M. R. Garey, and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[19] I. Spillinger, and G. M. Silberman, "Improving the Performance of a Switch-Level Simulator Targeted for a Logic Simulation Machine", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-5, No. 3 (July, 1986), pp. 396–404.

[20] R. E. Tarjan, "Fast Algorithms for Solving Path Problems", *J. ACM*, Vol. 23, No. 3 (July, 1981), pp. 594–614.

[21] E. C. Ogbuobiri, W. F. Tinney, and J. W. Walker, "Sparsity-Directed Decomposition for Gaussian Elimination on Matrices", *IEEE Trans. on Power Apparatus and Systems*, Vol. PAS-89, No. 1 (January, 1970), pp. 141–150.

[22] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, PhD Thesis, Univ. of California, Berkeley, Dept. of Elec. Eng. (1975).

[23] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized Nested Dissection", *SIAM Journal on Numerical Analysis*, Vol. 16, No. 2 (April, 1979), pp. 346–358.

[24] R. J. Duffin, "Topology of Series-Parallel Networks", *J. Math. Anal. and Applications*, Vol. 10 (1965), pp. 303–318.

[25] C. A. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[26] L. A. Glasser, and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley, 1985.

[27] N. H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1985.

[28] M. Annaratone, *Digital CMOS Circuit Design*, Kluwer Academic Publishers, 1986.

[29] M. Horowitz, private communication (1985).

[30] C. Lutz, S. Rabin, C. Seitz, and D. Speck, "Design of the MOSAIC Element," *Conf. on Advanced Research in VLSI*, MIT (1984), pp. 1–10.

[31] A. V. Aho, and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, 1972.

[32] A. V. Aho, and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, 1977.

[33] J. A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle", *J. ACM*, Vol. 12, No. 1 (1965).

[34] M. A. Breuer, and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976, p. 46.

[35] V. Pan, and J. Reif, "Efficient Parallel Solution of Linear Systems", Technical Report TR-02-85, Aiken Computation Laboratory, Harvard University, 1985.

[36] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. on Computers*, Vol. C-35, No. 8 (August, 1986) pp. 677–691.

Boolean Analysis of MOS Circuits*

Randal E. Bryant
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

February 6, 1987

Abstract

The switch-level model represents a digital metal-oxide semiconductor (MOS) circuit as a network of charge storage nodes connected by resistive transistor switches. The functionality of such a network can be expressed as a series of systems of Boolean equations. Solving these equations symbolically yields a set of Boolean formulas that describe the mapping from input and current state to the new network state. This analysis supports the same class of networks as the switch-level simulator MOSSIM II and provides the same functionality, including the handling of bidirectional effects and indeterminate (X) logic values. In the worst case, the analysis of an n node network can yield a set of formulas containing a total of $O(n^3)$ operations. However, all but a limited set of dense, pass-transistor networks give formulas with $O(n)$ total operations. The analysis can serve as the basis of efficient programs for a variety of logic design tasks, including: logic simulation (on both conventional and special purpose computers), fault simulation, test generation, and symbolic verification.

Keywords and phrases: switch-level networks, symbolic analysis, logic simulation, fault simulation, simulation accelerators.

1 Introduction

The switch-level model [1] has proved successful as an abstract representation of digital metal-oxide semiconductor (MOS) circuits for a variety of applications. This model represents a circuit in terms of its exact transistor structure but describes the electrical behavior in a highly idealized way. It expresses transistor conductances and node capacitances by discrete strength and size values; represents node voltages by discrete states 0, 1, and X (for invalid or indeterminate); and makes no attempt to model exact circuit timing. The switch-level model can capture many of the important phenomena encountered in MOS

*This research was supported in part by the Defense Advanced Research Projects, Agency ARPA Order Number 4976, and in part by the Semiconductor Research Corporation under Contract 36-01-068.

circuits such as: ratioed, complementary, and precharged logic; dynamic memory; and bidirectional pass transistors. Unlike programs that attempt to model circuits at a detailed electrical level, programs based on the switch-level model can operate at speeds approaching those of their counterparts based on more traditional gate-level models. Examples of applications that have successfully applied switch-level models include logic simulators [1,2], fault simulators [3,4], test pattern generators [5,6], and symbolic verifiers [7,8].

1.1 Switch-Level Algorithms

Most programs that model circuits at the switch level utilize totally different algorithms than those developed for logic gate circuits. To accommodate the bidirectional nature of the transistors, they compute the state of a node by applying graph algorithms to trace the connections between nodes formed by conducting transistors. This departure from tradition has several drawbacks. First, considerable effort is often required to adapt existing techniques for use at the switch-level. For example, in implementing the fault simulator FMOSSIM, we found it quite challenging to adapt concurrent simulation techniques [9], although the resulting performance proved well worth the effort. Similarly, automatic test pattern generation for switch-level circuits has not yet reached the success achieved for logic gate circuits. Second, although programs based on the switch-level model have reasonable performance, they fall short of those based on gate-level models. Computing node states by applying graph algorithms to the transistor data structure requires significantly greater effort than computing the output of a logic gate. Finally, these algorithms do not map well onto the special purpose processors that have been developed to accelerate such tasks as logic gate simulation [10,11,12]. Although special purpose processors for switch-level simulation have been designed and constructed [13,14], these processors require a fair amount of specialized hardware. It is unlikely they will ever achieve the cost/performance of processors that support only gate-level evaluation.

1.2 A New Approach

This paper proposes a new approach that deals with all aspects unique to the switch-level model in a preprocessing step. The preprocessor "compiles" a switch-level network into a set of Boolean formulas. For each node, a pair of formulas specifies its steady state response as a function of the initial node states. A simulator can then compute new node states by simply evaluating the appropriate formulas. Fault simulators and test generators can utilize traditional techniques by treating the set of formulas like a logic gate network. The formulas can be translated directly into machine language instructions for fast evaluation on a general purpose computer, or they can be mapped onto any special-purpose hardware that supports Boolean evaluation. An efficient symbolic analyzer, the subject of this paper, serves as the basis of this preprocessing.

This approach has advantages over traditional methods of switch-level evaluation in terms of both speed and flexibility. As an analogy, a programming language compiler yields a performance advantage over an interpreter, because the cost of translating the program into machine instructions is paid only once during compilation rather than repeatedly during execution. Similarly, the analyzer gives a performance advantage over traditional switch-level algorithms, because the added cost of switch-level evaluation is paid only once during preprocessing. In contrast to special purpose hardware for switch-level evaluation, many extensions to the model can be made by simply modifying the analyzer, a much simpler task than modifying the hardware. Furthermore, the Boolean description of switch-level subnetworks can more easily be combined with subnetworks modeled at other levels for mixed-mode evaluation. Finally, as will be discussed briefly, the preprocessor generates a description that can be executed with a far greater degree of parallelism than is possible with more conventional switch-level algorithms.

The analyzer described in this paper supports the same class of switch-level networks as the simulator MOSSIM II [1]. It captures all aspects of the MOSSIM II model including: bidirectional effects, different signal strengths, and indeterminate (X) logic values. The analysis of an n node network produces a set of formulas with a total of at most $O(n^3)$ operations. For all but a very small class of dense, pass transistor networks (e.g., barrel shifters), at most $O(n)$ operations are required. Hence, for practical purposes this approach incurs the same asymptotic complexity as other switch-level programs.

1.3 Related Work

Other researchers have developed preprocessors to translate a switch-level network into some algebraic representation that allows efficient evaluation. These previous efforts had, for the most part, limited generality and accuracy. In addition, they did not achieve acceptable efficiency. Pfister of IBM [15] probably deserves credit for originating the idea of describing arbitrary MOS circuits in terms of Boolean operations. He was seeking a way to perform switch-level simulation on the Yorktown Simulation Engine (YSE) [10].

Researchers at IBM [16] have modified and adapted a traditional switch-level algorithm for execution on the YSE. Their approach can be viewed as generating code to iteratively solve a system of equations in an algebra where elements encode both the strength and the state of a signal [17]. To accommodate the small word size of the machine, they restrict the number of signal strengths to 3 and use a pessimistic method for computing the effects of unknown states. More seriously, since the machine cannot perform data dependent branches, their code must always iterate a worst case number of times. For many transistor structures with n nodes and t transistors, this requires a total of $O(n\,t)$ YSE instructions, a high cost in both space and time. In a related effort, the SLS program developed at IBM [18] generates code for a general purpose computer that executes a single iteration in the solution of the same system of equations solved by MOSSIM II. During simulation this code is executed repeatedly until the values converge. Although this program achieves impressive performance on a variety of circuits, a significant class of pass transistor networks can require many iterations to converge. Furthermore, this approach cannot be implemented on existing simulation hardware, nor can it aid such tasks as test pattern generation or symbolic verification.

Others have attempted to express switch-level algorithms in terms of either Boolean or closely related algebras. All of these efforts have yielded highly inefficient results—in the worst case the size of the algebraic description can grow exponentially with the size of the network. These programs partition the circuit into subnetworks and analyze each subnetwork separately. Most subnetworks are quite small—containing no more than 10 transistors. Hence even an exponential algorithm can have practical value. However, we have often encountered circuits containing subnetworks of 1000 or more transistors. For such cases these algorithms would be totally inadequate. The method developed by Cerny and Gecsei [19] creates a symbolic representation of all possible partitionings of a subnetwork into connected components formed by the conducting transistors. All but the smallest subnetworks have many partitionings, and hence this approach has limited potential. The methods of Ditlow, *et al*, [20] of Hajj and Saab, [21] and of Terman [2] enumerate the set of all simple paths to each node and then encode information about each path algebraically. For many pass transistor networks, (e.g., the Tally circuit of Mead and Conway [22]), the number of such paths grows exponentially with the number of transistors. Furthermore, all of these methods place more restrictions on the class of networks than does MOSSIM II, and some do not do as well at modeling the effects of X values. Finally, the method of Hajj and Saab utilizes a mixed Boolean-integer algebra, in which the 1's in different sets of Booleans must be tabulated and compared. Such an algebra seems needlessly awkward and would be hard to implement on most simulation hardware.

In a different application of symbolic analysis, the MOSSYM program [7] simulates MOS circuits ***symbolically***. A symbolic simulator resembles a conventional simulator except that the input patterns may consist of Boolean variables in addition to the constants 0 and 1. The node states computed by the simulator represent Boolean functions over the present and past input variables. This program is designed to rigorously verify digital circuits, proving their correctness for all possible input sequences. As a consequence, it must solve Boolean equivalence, a well-known NP-hard problem [23]. The worst case performance of the program is exponential in the number of variables, and many researchers believe no better performance can be achieved. The methods used by MOSSYM for computing the Boolean behavior of a switch-level network form the basis of the analyzer described here. However, the analyzer can use different data structures and algorithms for representing Boolean functions, since it need not prove equivalence. Consequently, while the simplification algorithms may not yield the most compact formulas possible, they have acceptable worst case performance. Future versions of MOSSYM will operate on preprocessed networks rather than on the transistor structure directly, gaining the same benefits from preprocessing as do more conventional simulators.

1.4 Overview

The analyzer presented in this paper overcomes many weaknesses of the previous attempts. Important features include:

- It partitions the network into channel-connected subnetworks and derives the steady state response of each subnetwork separately. This partitioning divides the analysis task into smaller subproblems.

- It encodes logic states 0, 1, and X with pairs of Boolean values. By this encoding, it can accurately characterize the effects of unknown node and transistor states with Boolean formulas.
- Starting with the maximum strength level and working downward, it derives systems of Boolean equations for each strength level. It can capture the effects of any (fixed) number of signal strengths. These systems of equations express the effects of all paths in the graph but lend themselves to solution methods of polynomial complexity.
- It solves the equations symbolically by Gaussian elimination. Gaussian elimination can exploit the sparse structure of the networks to solve most n node subnetworks with $O(n)$ algebraic operations.
- It represents the set of Boolean formulas as a directed acyclic graph (DAG). This representation naturally allows sharing of common subexpressions. The size of the DAG describing the steady state response of all nodes in a subnetwork is bounded by the number of algebraic operations required during the Gaussian eliminations.

Comparing the analyzer to the inner workings of the circuit-level simulator SPICE [24] lends some insight into the underlying ideas. During transient analysis, SPICE computes the behavior of a nonlinear network at each time point by performing a series of iterations, each of which involves setting up a system of linear equations and solving it by Gaussian elimination. Similarly, our analyzer computes the behavior of a network of non-Boolean switches (due to the different signal strengths and the X states) by performing a series of iterations, each of which involves setting up 3 systems of Boolean equations and solving them by Gaussian elimination. In this respect, both programs apply the powerful mathematical technique of solving a difficult problem over a poorly structured domain by recasting it as a series of problems in a more tractable domain for which efficient, highly developed algorithms exist.

Unlike SPICE, however, the analyzer iterates in a fixed progression over signal strengths rather than until it reaches some convergence criterion. This progression is possible because of the discrete nature of signal strengths. At a given strength level, the analyzer has already computed the effects of stronger signals and can safely ignore weaker ones. Furthermore, rather than being performed on each time step, the analyzer need only compute the behavior once, yielding a set of formulas that are evaluated repeatedly during simulation. Such a symbolic analysis is possible because of the simpler natures of both the circuit elements and the mathematical domain. Whereas SPICE must linearize the network by evaluating complex device models at the current operating points of the circuit elements, the analyzer need only evaluate the effects of the possible paths at each strength level. Furthermore, Boolean formulas are far easier to manipulate and simplify than formulas over real numbers. Thus, while interesting parallels exist between SPICE and symbolic switch-level analysis, many factors contribute to make the latter far more efficient.

A companion paper [25] provides background on the mathematical and algorithmic techniques used in the analysis. This paper gives a detailed formulation of the switch-level model in terms of Boolean algebra. It also describes several extensions to the model, including ways to model circuit faults, degraded logic signals, and charge decay. These extensions demonstrate the power of the basic framework. The analyzer can incorporate new modeling features by modifying the basic systems of equations slightly. Future papers will cover implementation issues, applications, and experimental results.

The remainder of the paper is organized as two major parts. The first, consisting of Sections 2–4, formulates the behavior of a switch-level network as a system of equations in an abstract Boolean algebra. The second part, consisting of Sections 5–7, presents refinements of the technique, examples, and extensions to the switch-level model.

2 The Switch-Level Model

The switch-level model considered here has been described in detail elsewhere [1]. This section gives an overview of the model in terms of a cleaner notation and defines the symbolic analysis problem.

2.1 Network Model

A switch-level network consists of a set of nodes and a set of transistors. A node is classified as either input or storage. An *input* node represents a connection to a signal source external to the chip, supplying either power, ground, clock, or data. A *storage* node, like a capacitor in an electrical network, retains its state in the absence of applied inputs and can share charge with other storage nodes. The voltage on node n[1] is represented by its *state* $n \in \{0, 1, X\}$, with 0 and 1 corresponding to low and high voltage levels, respectively, and X corresponding to an indeterminate voltage between low and high indicating an uninitialized network state or an error condition caused by a short circuit or charge sharing.

A storage node has a characteristic *size* from the set $\{1, 2, \ldots, k\}$. This size indicates, in a highly simplified way, the node capacitance relative to that of other nodes with which it may share charge. That is, when a set of storage nodes share charge (due to connections by conducting transistors), only the connected nodes of maximum size determine the outcome. Input nodes are indicated by size $w > k$. The set $\mathcal{N}_s$ contains all nodes of size s, and hence $\mathcal{N}_w$ denotes the set of input nodes.

A transistor has terminals labeled, "gate", "source", and "drain". It acts as a resistive switch connecting the source and drain nodes controlled by the state of the gate node. Transistors act as bidirectional elements with no predetermined direction of information or current flow. A transistor has a *type* indicating the conditions under which it will become conducting. A *d-type* transistor always conducts; an *n-type* conducts when its gate has

[1]This presentation uses a notation where nodes are named by lower-case letters, e.g., m, n, their current states are indicated by italicized, lower-case letters, e.g., m, n, and their new states are indicated by italicized, upper-case letters, e.g., M, N.

state 1; while a *p-type* conducts when its gate has state 0. When the gate node of an n-type or p-type transistor has state X, the transistor can range between fully conducting and open circuited. Transistor states 0, 1, and X represent conduction levels nonconducting, fully conducting, and indeterminate, respectively.

Each transistor has a characteristic *strength* from the set $\{k+1, k+2, \ldots, w-1\}$. This strength indicates, in a highly simplified way, the transistor conductance relative to those of other transistors in a ratioed circuit. That is, every path of conducting transistors has a characteristic strength equal to the that of the weakest transistor in the path. When a set of paths form from several input nodes to a storage node, only those inputs connected by maximum strength paths determine the new node state. For nodes m and n, the set $\mathcal{T}_s(\mathtt{m}, \mathtt{n})$ contains all transistors of strength s having these two nodes as source and drain.

2.2 The Channel Graph

The *channel graph* represents the interconnection structure of a switch-level network. This graph has the storage nodes of the circuit as vertices, and an edge $(\mathtt{m}, \mathtt{n})$ for each pair of storage nodes m and n such that $\mathcal{T}_s(\mathtt{m}, \mathtt{n}) \neq \emptyset$ for some strength s. It describes the static (independent of transistor state) interconnections between storage nodes formed by the source-drain connections of the transistors. The channel graph is considered either undirected or directed depending on context. When talking about general structural properties of a circuit, an undirected graph simplifies the discussion. On the other hand, symbolic analysis requires a directed graph, because the labels assigned to the edges are direction sensitive.

In general, a channel graph consists of many connected components. Therefore, it defines a partitioning of the switch-level network into a set of *channel-connected subnetworks*, where each subnetwork consists of the set of nodes in a graph component, plus the set of transistors for which these nodes are sources or drains. Note that an input node is not part of any subnetwork, but a transistor for which the node is source (drain) is in the subnetwork of its drain (source) node.

Within a subnetwork, the behavior can be complex and difficult to analyze due to the bidirectional transistors and the many ways state forms in a MOS circuit. The interactions between subnetworks, however, are much more straightforward. Each subnetwork acts as a sequential logic element having as inputs the input nodes connected to transistor sources and drains as well as the gate nodes of the transistors. The subnetwork state is stored as charge on the storage nodes, and the outputs are those nodes that are gate nodes of transistors in other subnetworks. Hence, the overall operation of a switch-level simulator is similar to that of a logic gate simulator—changing values on the subnetwork inputs require updating the state and outputs, and these changing output values in turn affect other subnetworks. The challenge then is to develop formulas representing the behavior of individual subnetworks.

In practice, many subnetworks are small—containing at most 10 transistors. However, we have encountered subnetworks with over 5000 transistors (essentially the entire data path of a 16-bit microprocessor [26]), and hence the analysis of each subnetwork must be as efficient as possible.

2.3 Steady State Response

The steady state response function describes the behavior of a subnetwork. Informally, this function can be explained as follows. For a given set of connected input node and initial storage node states, the transistors are set according to their gate node states. The transistors in the 1 and X states create (potentially) conducting paths from input nodes to storage nodes and between pairs of storage nodes, causing the storage nodes to attain new voltage levels. The steady state response for a node equals the state (0, 1, or X) this node would attain if the transistors were held fixed long enough for the nodes to stabilize. When nodes or transistors in the X state are present, the steady state response on a node equals 0 or 1 only when it would attain this unique state regardless of the voltages and conductances of these nodes and transistors. Otherwise the steady state response equals X.

The steady state response for a subnetwork is defined formally in terms of the paths between nodes formed by the conducting transistors. This approach unifies the variety of different ways logic values form in MOS circuits including: stored charge, charge sharing, and both ratioed and complementary logic. For a given set of transistor states, transistors in the 1 and X state form connections between their source and drain nodes. A *rooted path* p is a directed path originating at node $Root(p)$, terminating at node $Dest(p)$ and consisting of a (possibly empty) set of transistors $Trans(p)$. The *strength* of path p, denoted $|p|$ is defined as

$$|p| \;=\; \min\left[Size[Root(p)],\ \min_{t\in Trans(p)} Strength(t)\right]$$

A rooted path represents a source of charge from its root to its destination with driving ability indicated by its strength. Rooted paths can be classified into three types according to their strength. A path with strength $1 \leq |p| \leq k$ represents a source of stored charge from a storage node with an approximate capacitance determined by the size of this node. Note that the stored charge initially on the node is represented by a path with root and destination equal to the node and with no transistors. A path with strength $k < |p| < w$ represents a source of current from an input node with an approximate conductance determined by the strength of the weakest transistor in the path. A path with strength $|p| = w$ must contain no edges and have an input node as both root and destination. Such a path represents the external current supplied to the input node. The overall ranking of path strengths reflects the fact that a connection from an input node can override any stored charge, while a direct connection to an input can override any resistive connection from some other input.

The steady state response of a node depends only on the paths to the node that are not "blocked". A *definite* path is defined as a rooted path p such that no transistor in $Trans(p)$ is in the X state. A path p is said to be *blocked* if for some initial segment p' of p (i.e. $Root(p') = Root(p)$ and $Trans(p') \subseteq Trans(p)$) and for some *definite* path q,

$Dest(p') = Dest(q)$ and $|p'| < |q|$. Intuitively, a path is blocked if the source of charge it represents would be overridden by a stronger source at some intermediate node. Define the path relation $\mathcal{P}$ between pairs of nodes as $\mathtt{m}\ \mathcal{P}\ \mathtt{n}$ when there is an unblocked path p with $Root(p) = \mathtt{m}$ and $Dest(p) = \mathtt{n}$. Then the steady state response on node n, denoted N, is given by the equation

$$N = \mathrm{lub}\{m | \mathtt{m}\ \mathcal{P}\ \mathtt{n}\}, \tag{1}$$

where "lub" represents the least upper bound over the ordering $0 < X$ and $1 < X$. In other words, if all unblocked sources of charge to a node drive it to 0 (or to 1), then the steady state response equals 0 (or 1). Otherwise, if the node is driven by conflicting sources or by sources of unknown value, the steady state response equals X. It can be shown that this characterization of the steady state response provides an accurate modeling of the effects of unknown states as well as several important mathematical properties [1].

2.4 State Encoding

To cast the switch-level model in terms of Boolean operations, a state value $y \in \{0, 1, X\}$ is encoded as two Boolean values $y.1, y.0 \in \{0, 1\}$ as follows

y	$y.1$	$y.0$
1	1	0
0	0	1
X	1	1

Formally, .1 and .0 are operators, expressed in postfix notation, mapping elements of $\{0, 1, X\}$ to elements of $\{0, 1\}$. The combination $y.1 = y.0 = 0$ does not represent a valid state. It can be considered a "don't care" combination in the derivation. With this encoding, if y is the least upper bound of a set A consisting of elements $a \in \{0, 1, X\}$, then

$$y.1 = \bigvee_{a \in A} a.1 \tag{2}$$

$$y.0 = \bigvee_{a \in A} a.0, \tag{3}$$

where $\bigvee$ denotes the Boolean sum of a set of elements.

With this Boolean encoding of state values, the symbolic analysis problem can be defined as follows. For each node n, introduce Boolean variables $n.1$ and $n.0$ to represent the encoded value of the initial node state. Of course, when the node is known to have a fixed state (e.g., power or ground), its state can be encoded by constants rather than variables. For each node n, we are to derive Boolean formulas, denoted $N.1$ and $N.0$, for the encoded steady state response in terms of the node state variables. The encoding of node states makes it possible to express the three-valued circuit behavior using conventional Boolean algebra. This greatly simplifies the algebraic manipulation portion of the symbolic analyzer, at the cost of requiring a pair of formulas to describe each node.

In terms of this encoding, Equations 2 and 3 can be applied to Equation 1 to give:

$$N.1 \quad = \quad \bigvee_{m \mathcal{P} n} m.1 \tag{4}$$

$$N.0 \quad = \quad \bigvee_{m \mathcal{P} n} m.0. \tag{5}$$

3 Mathematical and Algorithmic Background

This section briefly summarizes the mathematical notation, results, and algorithms developed in the companion paper.

3.1 Symbolic Algebra

A Boolean formula describes a function mapping each possible combination of values for the set of p variables to 0 or 1. Mathematically, symbolic analysis can be viewed as manipulating elements of the algebra $\langle \mathcal{B}, \wedge, \vee, \neg, \mathbf{0}, \mathbf{1} \rangle$, where

$$\mathcal{B} \quad = \quad \left\{ f\colon \{0,1\}^p \rightarrow \{0,1\} \right\}.$$

The operations $\wedge$, $\vee$, and $\neg$ denote Boolean AND, OR, and NOT, respectively, applied to functions. The distinguished elements $\mathbf{0}$ and $\mathbf{1}$ represent the constant functions that yield 0 and 1, respectively, for all argument values. This process of *abstracting* from a primitive domain to one of functions, while maintaining the algebraic properties, forms the basis of symbolic analysis.

The Boolean product of the elements in a set A is denoted $\bigwedge_{a \in A} a$. The product of an empty set is defined to equal $\mathbf{1}$. Similarly, the Boolean sum of the elements in a set A is denoted $\bigvee_{a \in A} a$. The sum of an empty set is defined to equal $\mathbf{0}$.

3.2 Systems of Boolean Equations

Systems of Boolean equations provide a mathematical formalism for networks of switches much as do systems of linear equations for networks of resistors. However, to emphasize the sparse nature of the networks, labeled graphs are preferred to a matrix notation.

A system of Boolean equations is represented by an edge and vertex labeling on a directed graph (V, E), where a labeling indicates of an assignment of elements of $\mathcal{B}$ to every edge or vertex. The system $[A, b]$ consists of an edge label $A(u,v) \in \mathcal{B}$ for each $(u,v) \in E$ and a vertex label $b(v) \in \mathcal{B}$ for each $v \in V$. Vertex labeling x *satisfies* the system $[A, b]$ when

$$x(v) \quad = \quad b(v) \vee \bigvee_{(u,v) \in E} [x(u) \wedge A(u,v)]$$

for every $v \in V$. In general, many labelings may satisfy a system, but by defining an appropriate partial ordering of the elements of $\mathcal{B}$, every system can be shown to have a unique minimum satisfying labeling. This labeling is termed the system *solution*.

The solution of a Boolean system describes the conditions under which conducting paths will form in a switch network. More precisely, $P_{u,v}$ is defined as the set of all paths from vertex u to v in the graph. For solution x of the system $[A, b]$:

$$x(v) = \bigvee_{u \in V} \bigvee_{p \in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t) \in p} A(s,t) \right]$$

for all vertices v. In other words, if the "value" of a path is defined as the Boolean product of the initial vertex label and all edge labels, then $x(v)$ equals the Boolean sum of the values of all paths terminating at v.

A *dual* system of Boolean equations is similar to a normal system, but with the roles of $\wedge$ and $\vee$ interchanged. Labeling x satisfies the dual system $[A, b]^D$ when

$$x(v) = b(v) \wedge \bigwedge_{(u,v) \in E} [x(u) \vee A(u,v)]$$

for every $v \in V$. Dual systems express conditions under which conducting paths are absent in a switch network. More precisely, let $\overline{A}$ denote the edge labeling with each element equal to the Boolean complement of the corresponding element of A, and similarly for $\overline{b}$. For solution x of the dual system $[\overline{A}, \overline{b}]^D$:

$$x(v) = \neg \bigvee_{u \in V} \bigvee_{p \in P_{u,v}} \left[b(u) \wedge \bigwedge_{(s,t) \in p} A(s,t) \right]$$

Gaussian elimination can solve a system of Boolean equations (either normal or dual), with operations $\wedge$ and $\vee$ replacing the real arithmetic used when solving linear systems. Most channel graphs fall into a class called General Series-Parallel (GSP). This class includes both conventional series-parallel graphs as well as ones containing acyclic branches. Gaussian elimination requires at most $12n$ algebraic operations to solve a Boolean system defined over an n vertex GSP graph.

3.3 Boolean Formula Representation

A directed acyclic graph (DAG), with leaves denoting variables and constants and with nodes denoting Boolean operations, can represent a set of Boolean formulas. The symbolic solution of a Boolean system generates a DAG with each formula indicated by a pointer to some DAG node. The symbolic manipulator applies a Boolean operation to two formulas by creating a new node with branches to the nodes representing the arguments. By this means, the total size of the Boolean description generated is bounded by the number of algebraic operations performed during Gaussian elimination. The manipulator applies graph transformation rules corresponding to the laws of Boolean algebra to simplify the formulas, thereby reducing the DAG size.

A given Boolean function has many different DAG representations. The exact formula structure generated during Gaussian elimination depends on the order in which vertices are eliminated. Hence, the result is neither unique nor of minimum size. Using only these "weak" symbolic manipulation algorithms, however, avoids trying to solve any NP-hard problems.

4 Boolean Representation of the Steady State Response

This section formulates the steady state response of a node in terms of Boolean operations as well as relations and predicates between the nodes. It then develops systems of Boolean equations to describe the steady state response symbolically.

4.1 Strength Encoding

The analyzer accounts for the effects of different strength signals by starting at the maximum strength and working downward, each time adding in the effects from paths of the next lower strength. This approach captures signal strength effects in the *structure* of the equations to be solved. In contrast, MOSSIM II and most other switch-level simulators encode strength effects in the *algebra* in which the equations are expressed. This structural approach makes it possible to express the behavior in terms of Boolean algebra. It has the disadvantage that the number of equations to be solved is proportional to the total number of signal strengths w, whereas the algebraic approach can use algorithms with complexity essentially independent of w. This does not compromise the efficiency significantly, however, because few MOS circuits require more than 6 signal strengths to characterize their behavior (2 storage node sizes, 3 transistor strengths, and 1 input node size.)

For a given set of transistor states and for signal strength s, the path relation $\mathcal{P}_s$ is defined as $\mathtt{m}\ \mathcal{P}_s\ \mathtt{n}$ when there exists an unblocked path of strength greater than or equal to s from $\mathtt{m}$ to $\mathtt{n}$. If we define $N.1_s$ as

$$N.1_s \quad = \quad \bigvee_{\mathtt{m}\mathcal{P}_s\mathtt{n}} m.1 \tag{6}$$

and $N.0_s$ as

$$N.0_s \quad = \quad \bigvee_{\mathtt{m}\mathcal{P}_s\mathtt{n}} m.0 \tag{7}$$

then $N.1_s$ (respectively, $N.0_s$) describes the conditions under which node $\mathtt{n}$ will be the destination of an unblocked path of strength s or greater originating at a node $\mathtt{m}$ with $m = 1$ or X, (resp., 0 or X.) By this definition, $N.1 = N.1_1$ and $N.0 = N.0_1$ for node $\mathtt{n}$.

Consider the general form of an unblocked path from node $\mathtt{m}$ to node $\mathtt{n}$ having strength greater than or equal to s. It can be an unblocked path of strength greater than s, in which case $\mathtt{m}\ \mathcal{P}_{s+1}\ \mathtt{n}$. Otherwise, the path must have strength s, the possible forms of which are illustrated in Figure 1. For a driving path, node $\mathtt{m}$ must be an input node, and the path must consist of a (possibly empty) sequence of transistors of strength greater than s to some node $\mathtt{l}$, followed by a transistor of strength s, followed by a (possibly empty) sequence of transistors of strength greater than or equal to s to $\mathtt{n}$. The portion from $\mathtt{m}$ to $\mathtt{l}$ cannot be blocked, and hence $\mathtt{m}\ \mathcal{P}_{s+1}\ \mathtt{l}$. Furthermore, no node in the portion from $\mathtt{l}$ to $\mathtt{n}$, except $\mathtt{l}$, can be the destination of a definite path of strength greater than s. For a charging path, node $\mathtt{m}$ must be a storage node of size s, and the path must consist of a

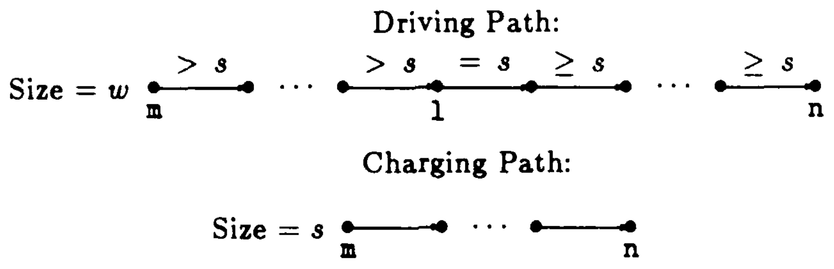


Figure 1: **General Form of a Strength s Path.** For $s > k$, the path originates at an input node, passes through a strength s transistor, and contains no weaker transistors. For $s \leq k$, the path originates at a storage node of size s.

sequence of transistors from m to n such that no node is the destination of a definite path of strength greater than s.

These conditions can be incorporated into a formal definition of $\mathcal{P}_s$ by introducing additional predicates and relations. The conditions expressed by these conditions and relations can then be expressed symbolically as the solutions to systems of Boolean equations. For each node n define the predicate $\mathcal{C}_s(\mathrm{n})$ as holding when n is *not* the destination of any definite path of strength greater than or equal to s. This predicate expresses the condition that the node is *clear*, i.e., not blocked, for signals of strength $s-1$. Define the relation $\mathcal{Q}_s$ as m $\mathcal{Q}_s$ n when the following conditions hold:

- There is a path p in the network with $Root(p) = \mathrm{m}$ and $Dest(p) = \mathrm{n}$ consisting only of transistors with state 1 or X and strength greater than or equal to s.
- $\mathcal{C}_{s+1}(\mathrm{l})$ holds for every node l in p other than m.

The relation $\mathcal{P}_s$ can then be expressed as $\mathcal{P}_w = \{(\mathrm{n},\mathrm{n}) | \mathrm{n} \in \mathcal{N}_w\}$, and for $s < w$:

$$\begin{aligned}\mathcal{P}_s &= \mathcal{P}_{s+1} \cup \left\{(\mathrm{m},\mathrm{n}) \middle| \exists \mathrm{l},\ \mathrm{m}\ \mathcal{P}_{s+1}\ \mathrm{l} \text{ and } \mathrm{l}\ \mathcal{Q}_s\ \mathrm{n}\right\} \\ &\quad \cup \left\{(\mathrm{m},\mathrm{n}) \middle| \mathrm{m}\ \mathcal{Q}_s\ \mathrm{n},\ \mathcal{C}_{s+1}(\mathrm{m}), \text{ and } \mathrm{m} \in \mathcal{N}_s\right\}\end{aligned}$$

The three terms in this equation represent the three classes of unblocked paths having strength greater than or equal to s discussed above.

Substituting for the definition of $N.1_s$ gives

$$N.1_s = \bigvee_{\mathrm{m}\mathcal{P}_{s+1}\mathrm{n}} m.1 \vee \bigvee_{\mathrm{m}\mathcal{P}_{s+1}\mathrm{l}} \bigvee_{\mathrm{l}\mathcal{Q}_s\mathrm{n}} m.1 \vee \bigvee_{\mathrm{m}\mathcal{Q}_s\mathrm{n}} \mathcal{C}_{s+1}(\mathrm{m}) \wedge (\mathrm{m} \in \mathcal{N}_s) \wedge m.1$$

By Equation 6 the first term in this equation equals $N.1_{s+1}$. The second term can be transformed by reversing the ordering of the summation, applying Equation 6 inside the summation, and changing the summation variable as follows:

$$\bigvee_{\mathrm{m}\mathcal{P}_{s+1}\mathrm{l}} \bigvee_{\mathrm{l}\mathcal{Q}_s\mathrm{n}} m.1 = \bigvee_{\mathrm{l}\mathcal{Q}_s\mathrm{n}} \left(\bigvee_{\mathrm{m}\mathcal{P}_{s+1}\mathrm{l}} m.1 \right) = \bigvee_{\mathrm{l}\mathcal{Q}_s\mathrm{n}} L.1_{s+1} = \bigvee_{\mathrm{m}\mathcal{Q}_s\mathrm{n}} M.1_{s+1}.$$

The equation for $N.1_s$ then becomes

$$N.1_s \quad = \quad N.1_{s+1} \ \vee \ \bigvee_{\mathbf{m} \mathcal{Q}_s \mathbf{n}} \left(M.1_{s+1} \vee \left[\mathcal{C}_{s+1}(\mathbf{m}) \wedge (\mathbf{m} \in \mathcal{N}_s) \wedge m.1 \right] \right). \qquad (8)$$

By similar reasoning, $N.0_s$ can be written as

$$N.0_s \quad = \quad N.0_{s+1} \ \vee \ \bigvee_{\mathbf{m} \mathcal{Q}_s \mathbf{n}} \left(M.0_{s+1} \vee \left[\mathcal{C}_{s+1}(\mathbf{m}) \wedge (\mathbf{m} \in \mathcal{N}_s) \wedge m.0 \right] \right). \qquad (9)$$

Thus the steady state response at strength s is expressed in terms of the response at strength $s+1$, the relation $\mathcal{Q}_s$ and the predicate $\mathcal{C}_{s+1}$. Equations for both $\mathcal{Q}_s$ and $\mathcal{C}_{s+1}$ can be formulated as systems of Boolean equations, thereby formulating the steady state response in Boolean terms. Interestingly, this derivation yields a result similar to one derived by Byrd, Hachtel, and Lightner [27] based on an "order of magnitude" linear network model. They show that at each strength level, the effect of all stronger signals to a node can be represented by a single voltage source with voltage corresponding to the net signal value, while all weaker signals can be ignored. Terms of the form $M.1_{s+1}$ and $M.0_{s+1}$ in Equations 8 and 9 are analogous to a source at node m representing the net effect of the stronger signals at this node. Furthermore, these equations contain no terms representing signals of strength less than s.

4.2 Symbolic Formulation

With this background, we are ready to formulate the steady state response in terms of systems of Boolean equations. For a transistor t, the formulas *indefinite*(t) and *potential*(t) indicate whether the transistor is not definitely conducting (in state 0 or X) or potentially conducting (in state 1 or X) depending on the state of its gate node n as follows:

type	*indefinite*(t)	*potential*(t)
n-type	$n.0$	$n.1$
p-type	$n.1$	$n.0$
d-type	**0**	**1**

Let (V, E) be a directed graph corresponding to a single component of the channel graph. For each strength level s, such that $w > s \geq 1$, the analyzer sets up and solves three systems of Boolean equations (two normal and one dual) for different labelings of this graph. The solutions yield formulas for $N.1_s$, $N.0_s$, and $\mathcal{C}_{s+1}(\mathrm{n})$ for each storage node n in the subnetwork. In each case, the edge labeling describes the transistor connections between pairs of storage nodes, while the vertex labeling describes a combination of the initial value on the storage node plus those on input nodes connected by single transistors. This approach takes advantage of the fact that any network path passing through an input node must be blocked, and hence transistors connected to input nodes act as unidirectional switches.

Starting with $N.1_w = \mathbf{0}$ for any storage node n, the analyzer computes formulas for $N.1_s$, $w > s \geq 1$ by solving the system $[Conduct_s, init1_s]$. This system of equations is based on Equation 8. The edge labeling $Conduct_s$ expresses the conditions under which a sequence of transistors satisfies the conditions of the relation $\mathcal{Q}_s$:

$$Conduct_s(\mathtt{m},\mathtt{n}) \;=\; N.c_{s+1} \wedge \left[Conduct_{s+1}(\mathtt{m},\mathtt{n}) \vee \bigvee_{t \in T_s(\mathtt{m},\mathtt{n})} potential(t) \right], \tag{10}$$

where $Conduct_w(\mathtt{m},\mathtt{n}) = \mathbf{0}$. The term $N.c_{s+1}$ is a formula that symbolically expresses the predicate $C_{s+1}(\mathtt{n})$, as will be defined shortly. Note the asymmetry in the above edge labeling, where in general $Conduct_s(\mathtt{m},\mathtt{n}) \neq Conduct_s(\mathtt{n},\mathtt{m})$. It arises from the requirement that $C_{s+1}(\mathtt{l})$ must hold for all nodes l in the path *other* than the first one. By forming an edge label equal to the Boolean product of the conduction condition for the corresponding transistors and the clear condition for the edge destination, any path formed by these edges must be clear at all but the first node. The vertex labeling $init1_s$ combines terms inside the summation of Equation 8 for a storage node and for connected input nodes:

$$init1_s(\mathtt{n}) \;=\; \begin{cases} N.1_{s+1} \vee \left(N.c_{s+1} \wedge \bigvee_{\mathtt{m} \in \mathcal{N}_w} \bigvee_{t \in T_s(\mathtt{m},\mathtt{n})} [potential(t) \wedge m.1] \right) & s > k \\ N.1_{s+1} \vee (N.c_{s+1} \wedge n.1) & \mathtt{n} \in \mathcal{N}_s \\ N.1_{s+1} & \text{else} \end{cases} \tag{11}$$

Starting with $N.0_w = \mathbf{0}$, the analyzer computes formulas for $N.0_s$, $w > s \geq 1$ by solving the system $[Conduct_s, init0_s]$. This system is based on Equation 9. The edge labeling $Conduct_s$ is the same as before (Equation 10.) Vertex labeling $init0_s$ is analogous to $init1_s$:

$$init0_s(\mathtt{n}) \;=\; \begin{cases} N.0_{s+1} \vee \left(N.c_{s+1} \wedge \bigvee_{\mathtt{m} \in \mathcal{N}_w} \bigvee_{t \in T_s(\mathtt{m},\mathtt{n})} [potential(t) \wedge m.0] \right) & s > k \\ N.0_{s+1} \vee (N.c_{s+1} \wedge n.0) & \mathtt{n} \in \mathcal{N}_s \\ N.0_{s+1} & \text{else} \end{cases} \tag{12}$$

Finally, starting with $N.c_w = \mathbf{1}$ for any storage node n, the analyzer computes formulas for $N.c_s$, $w > s > 1$ by solving the dual system $[Indef_s, initc_s]^D$. This formula symbolically encodes the predicate $C_s(\mathtt{n})$. The computation is formulated as a dual system to express the absence of blocking paths. The edge labeling $Indef_s$ describes the conditions under which two storage nodes are *not* connected by a transistor in the 1 state of strength greater than or equal to s:

$$Indef_s(\mathtt{m},\mathtt{n}) \;=\; Indef_{s+1}(\mathtt{m},\mathtt{n}) \wedge \bigwedge_{t \in T_s(\mathtt{m},\mathtt{n})} indefinite(t), \tag{13}$$

where $Indef_w(\mathtt{m},\mathtt{n}) = \mathbf{1}$. The vertex labeling $initc_s$ indicates the conditions under which a storage node neither has size s, nor is the destination of a definite path of strength greater than s, nor is connected to an input node by a transistor with state 1 and strength s:

$$initc_s(\mathrm{n}) = \begin{cases} N.c_{s+1} \wedge \bigwedge_{\mathrm{m}\in\mathcal{N}_w} \bigwedge_{t\in T_s(\mathrm{m},\mathrm{n})} indefinite(t) & s > k \\ 0 & \mathrm{n} \in \mathcal{N}_s \\ N.c_{s+1} & \text{else} \end{cases} \tag{14}$$

To summarize, the computation of the steady state response formulas starts with $s = w - 1$ and works downward to $s = 1$. At each strength level the analyzer sets up and solves equations to compute $N.1_s$ and $N.0_s$ for every node. It then sets up and solves a dual system to compute $N.c_s$ for every node for use at the next lower strength level. The desired results for node n equal $N.1_1$ and $N.0_1$, respectively. Although the above presentation used names subscripted by s to represent the terms at different strength levels, the implementation need only retain the terms for the current strength as it iterates.

5 Refinements

The analyzer as described so far achieves good asymptotic performance in terms of the size of the formulas generated. However, the performance can be further improved by reducing the constant of proportionality, by generating a hierarchical description, or by maximizing the potential concurrency of the evaluation.

5.1 Nonessential Node Elimination

Until now, the presentation has assumed that the analyzer must compute the steady state response for every node in a subnetwork, as is done by most switch-level simulators. However, some nodes serve only as interconnection points in a circuit—they neither control any transistors nor form part of the circuit memory. For example, all intermediate nodes in the pullup and pulldown networks of nMOS and CMOS logic gates serve only as interconnections. For modeling circuit behavior, a program such as a simulator need only keep track of the states of "essential" nodes, i.e., those that can either directly or indirectly affect the value of a subnetwork output.

Once the analyzer has generated the DAG for a subnetwork, a postprocessor can prune it to include only those parts required to compute the states of essential nodes as follows. The postprocessor starts by marking the DAG nodes representing all formulas $N.1$ and $N.0$ for which n is a primary output of the circuit or is the gate of an n-type or p-type transistor. It then traces down the DAG and marks their descendants. If it encounters a leaf representing variable $m.1$ (respectively, $m.0$), and the DAG node representing the formula $M.1$ (respectively, $M.0$) has not been marked, then it marks this node and traces the descendants. This process continues until it can reach no further DAG nodes. The pruned DAG consists of those parts that have been marked.

Note that the degree of pruning depends on the degree to which the original formulas have been simplified, because simplification will typically reduce the number of variables occurring in a formula. Therefore, Boolean simplification tends to have a multiplicative effect—greater simplification reduces the size of the original DAG and also increases the amount by which it can be pruned.

5.2 Hierarchical Analysis

The presentation has also assumed that the analyzer must extract the function of every subnetwork in a circuit. However, most VLSI circuits contain repeated structures, and therefore many isomorphic subnetworks. A more efficient method would analyze only unique subnetworks. It would then produce a hierarchical representation in which each subnetwork instance references the appropriate Boolean description with its own set of node parameters. This hierarchical analysis would require less time and produce a more compact description. Such an approach, however, requires a method to recognize isomorphic subnetworks.

A circuit described hierarchically already has much of the commonality represented explicitly. Unfortunately, the hierarchical partitioning of the circuit will not, in general, conform to the partitioning required for symbolic analysis. That is, the circuit elements in a single channel-connected subnetwork may be declared in several components of the hierarchical description. To exploit this hierarchy, the analyzer must first modify the circuit description to respect subnetwork boundaries. It can do this by "pulling" all node and transistor specifications for each subnetwork up the hierarchy into the least common ancestor of the components in which they originally occurred.

Alternatively, the analyzer can extract common subnetworks by applying graph isomorphism techniques. Although efficient and reliable algorithms for general graph isomorphism have not yet been developed, heuristic methods developed in the context of interconnect verification have proved very successful [32,33]. Furthermore, if the analyzer fails to recognize some isomorphisms, the output will not be as compact, but the results will still be valid.

5.3 Maximizing Potential Parallelism

In the near future, preprocessors for switch-level networks will routinely generate code for computers that support high degrees of parallelism. For example, the YSE can have up to 256 processors operating simultaneously and communicating through a cross-bar switch. Under such conditions, it is more important for the preprocessor to reduce sequential constraints imposed by data dependencies rather than to minimize the formula size. Reducing data dependencies also simplifies scheduling on highly pipelined processors. As is mentioned in the companion paper, pivots can be chosen for Gaussian elimination such that the analysis of an n node, general series-parallel network yields a set of formulas with $O(n)$ total operations and maximum depth $O(\log n)$. Thus, given sufficient parallel resources, the steady state response for a subnetwork could be computed in sublinear time. In contrast, the algorithms used by MOSSIM II and all other switch-level simulators cannot achieve sublinear performance regardless of the processing capabilities. For example, they would be effectively limited to sequential execution when propagating a signal down a long chain of transistors. Gaussian elimination removes this constraint by collecting information about the entire chain and then redistributing the results, each time through expression trees of logarithmic depth. Thus a preprocessor based on Gaussian elimination becomes especially attractive for highly parallel systems.

6 Examples

This section highlights some characteristics of the analyzer by evaluating how the analysis would proceed for several general MOS implementations of logic gates, and by executing the algorithm on a small CMOS circuit. For circuits containing more than a handful of transistors, it becomes impractical to trace the execution steps in detail, and the resulting formulas are too large to examine manually. In studying these examples, the reader must keep in mind that the true strength of the analyzer lies in its ability to handle much larger circuits.

The presentation expresses performance by a parameter α, defined as the the total number of binary Boolean operations in the formulas divided by the number of transistors in the network being analyzed. Lower values of α indicate a more efficient analysis. Although this parameter only measures the size of the analyzer output, it also provides a reasonable indication of the time required for execution. A worst case analysis shows that α cannot exceed 240 for circuits with at most 6 signal strengths where all subnetworks have general series-parallel channel graphs [25]. This analysis, however, is far too pessimistic. Experiments on actual circuits indicate a typical range of 2 to 10.

6.1 General Logic Gates

Figure 2 illustrates the switch-level representations of three classes of MOS logic gates. In this figure the network N represents a pulldown network of n-type transistors. When viewed as a two terminal network of switches with control variables $a_1, \ldots, a_k$, the conditions under which a path forms across the terminals is given by its *transmission function* $T(a_1, \ldots, a_k)$. Similarly the network $\mathtt{N}_D$ represents a pullup network of p-type transistors. When viewed as a network of *positive* switches this network has a transmission function $T_D(a_1, \ldots, a_k)$ equal to the *dual* of T. That is, these two functions are related as

$$T(a_1, \ldots, a_k) = \neg T_D(\neg a_1, \ldots, \neg a_k).$$

Note also that T is the dual of T_D. In most cases, $\mathtt{N}$ is a series-parallel network with $\mathtt{N}_D$ its dual. That is, parallel connections in $\mathtt{N}$ correspond to series connections in $\mathtt{N}_D$, and *vice-versa*. However, these conditions are not mandatory—network duality is a sufficient, but not necessary, condition for functional duality. The formula obtained by solving a system of equations representing network $\mathtt{N}$ is equivalent to that obtained by solving a dual system of equations representing network $\mathtt{N}_D$, and *vice-versa*.

A static nMOS gate consists of a pulldown network of n-type transistors connecting the output to ground, and a weaker, depletion mode transistor connecting the output to

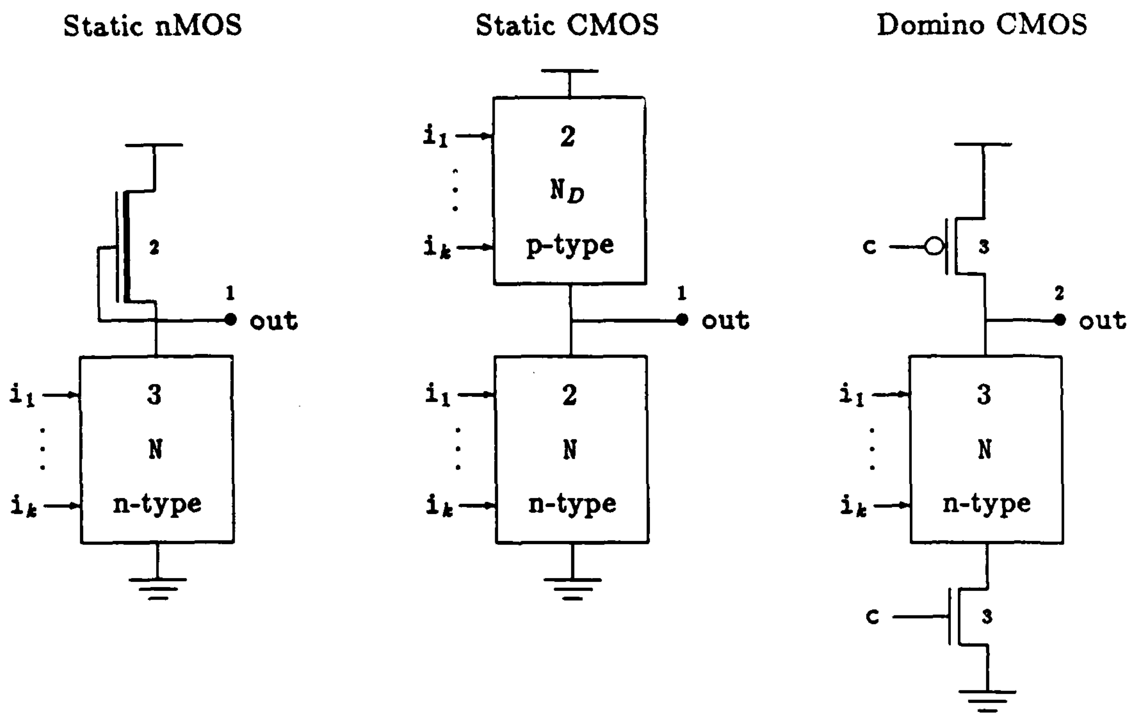


Figure 2: **Switch-level Representations of Logic Gates.** The boxes indicate networks of transistors labeled by strength and type. Transistors are labeled by their strength, and storage nodes by their size.

a	$a.1$	$a.0$	b	$b.1$	$b.0$	$Out.1$	$Out.0$	Out
0	0	1	0	0	1	1	0	1
0	0	1	1	1	0	1	0	1
0	0	1	X	1	1	1	0	1
1	1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	1	0
1	1	0	X	1	1	1	1	X
X	1	1	0	0	1	1	0	1
X	1	1	1	1	0	1	1	X
X	1	1	X	1	1	1	1	X

Table 1: **Three-valued Behavior of NAND Gate.** The formulas generated by the analyzer predict the same behavior as conventional ternary logic.

power. The storage node sizes make no difference in the result and are set to 1 for the example. The analysis of the steady state response at node out proceeds as follows:

$$\begin{aligned}
Out.1_3 &= \mathbf{0} \\
Out.0_3 &= T(i_1.1, \ldots, i_k.1) \\
Out.c_3 &= T_D(i_1.0, \ldots, i_k.0) \\[1em]
Out.1_2 &= Out.c_3 \wedge \mathbf{1} \quad = T_D(i_1.0, \ldots, i_k.0) \\
Out.0_2 &= T(i_1.1, \ldots, i_k.1) \\
Out.c_2 &= \mathbf{0} \\[1em]
Out.1_1 &= T_D(i_1.0, \ldots, i_k.0) \\
Out.0_1 &= T(i_1.1, \ldots, i_k.1)
\end{aligned}$$

giving $Out.1 = T_D(i_1.0, \ldots, i_k.0)$ and $Out.0 = T(i_1.1, \ldots, i_k.1)$. That is, the formula for $Out.0$ would express the function T applied to variables $i_1.1, \ldots, i_k.1$, while the formula for $Out.1$, arising from the dual analysis of network N, would express the function T_D applied to $i_1.0, \ldots, i_k.0$.

For example, a NAND gate with inputs a and b would have steady state response functions:

$$\begin{aligned}
Out.1 &= a.0 \vee b.0 \\
Out.0 &= a.1 \wedge b.1
\end{aligned}$$

Evaluating these formulas for all possible values of the variables yields the functionality shown in Table 6.1. As can be seen, these formulas capture the conventional ternary extension of the NAND function. In general, the pairs of formulas for all nMOS logic gates express a ternary behavior equal to the ternary extension of the corresponding gate function [?].

Node out is always the destination of a definite path of strength 2, and hence the stored charge of the nodes within N cannot affect its steady state response. Eliminating nonessential variables would then reduce the set of formulas to those expressing the steady state response of out. For the case of a series-parallel network, and an analysis by Gaussian elimination with node out eliminated last, the size of the formula generated will equal the number of transistors minus 1. This gives a performance measure α slightly less than 2. Even for networks that are not strictly series-parallel, such as ones containing bridges, α will not significantly exceed 2.

A static CMOS gate consists of a pulldown network of n-type transistors connecting the output to ground and a pullup network of p-type transistors connecting the output to power, where the two networks have dual transmission functions. Neither the transistor strengths nor the storage node sizes affect the gate function. For the example they are set to 2 and 1, respectively. The analysis of the steady state response at node out proceeds as follows:

$$\begin{aligned}
Out.1_2 &= T_D(i_1.0,\ldots,i_k.0)\\
Out.0_2 &= T(i_1.1,\ldots,i_k.1)\\
Out.c_2 &= T(i_1.1,\ldots,i_k.1)\wedge T_D(i_1.0,\ldots,i_k.0)
\end{aligned}$$

$$\begin{aligned}
Out.1_1 &= T_D(i_1.0,\ldots,i_k.0)\vee[x\wedge Out.c_2] &&= T_D(i_1.0,\ldots,i_k.0)\\
Out.0_1 &= T(i_1.1,\ldots,i_k.1)\vee[y\wedge Out.c_2] &&= T(i_1.1,\ldots,i_k.1)
\end{aligned}$$

where the terms x and y express the effects of the initial stored charge on nodes out and those internal to N and N_D. These terms are eliminated by absorption and hence are not shown in detail. The steady state response is therefore identical to that obtained for an equivalent nMOS gate even when some inputs equal X.

Unlike nMOS gates, there may be no definite driving path to out, and hence the paths representing sources of stored charge to out may not be blocked. However, in all such cases the gate output will equal X, and hence these paths have no effect. Unfortunately, the analyzer may not recognize the possible absorption of terms representing sources of stored charge by those representing driving paths. To do so, it must recognize that the formulas generated during the normal analysis of N and N_D are equivalent to those generated during the dual analysis of N_D and N, respectively. Using Gaussian elimination where node out is eliminated last, and the formula manipulation techniques described in the companion paper, these equivalences will be recognized for the most common case of N being series-parallel and N_D its dual. If these absorption conditions are recognized, then nonessential variable elimination will reduce the set of formulas to those representing the steady state response of out. Therefore, for series-parallel networks, the analyzer has a performance with α slightly less than 1. On the other hand, it will not do as well for networks that are not series-parallel, nor where N and N_D are not dual networks. Such cases are sufficiently rare to have little impact on the overall performance.

A domino CMOS gate [29,30] consists of a p-type precharge transistor, a pulldown network of n-type transistors, and an n-type discharge transistor that can connect the gate output to ground through the pulldown network. Both the precharge and the discharge transistors are gated by a common clock c. Transistor strengths do not affect the gate function and are set to 3 for the example. However, when connected in domino fashion, the output node must have greater capacitance than the internal nodes of N, because it may share charge with them. Therefore node out has size 2 and the nodes internal to N have size 1. The analysis of the steady state response at node out would proceed as follows:

$$\begin{aligned}
Out.1_3 &= c.0 \\
Out.0_3 &= T(i_1.1,\ldots,i_k.1) \wedge c.1 \\
Out.c_3 &= [T_D(i_1.0,\ldots,i_k.0) \vee c.0] \wedge c.1 \\
&= [T_D(i_1.0,\ldots,i_k.0) \wedge c.1] \vee [c.1 \wedge c.0] \\
\\
Out.1_2 &= c.0 \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.1] \vee [c.1 \wedge c.0 \wedge out.1] \\
Out.0_2 &= [T(i_1.1,\ldots,i_k.1) \wedge c.1] \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.0] \vee [c.1 \wedge c.0 \wedge out.0] \\
Out.c_2 &= 0 \\
\\
Out.1_1 &= c.0 \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.1] \vee [c.1 \wedge c.0 \wedge out.1] \\
Out.0_1 &= [T(i_1.1,\ldots,i_k.1) \wedge c.1] \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.0] \vee [c.1 \wedge c.0 \wedge out.0]
\end{aligned}$$

giving final results

$$\begin{aligned}
Out.1 &= c.0 \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.1] \vee [c.1 \wedge c.0 \wedge out.1] \\
Out.0 &= [T(i_1.1,\ldots,i_k.1) \wedge c.1] \vee [T_D(i_1.0,\ldots,i_k.0) \wedge c.1 \wedge out.0] \vee [c.1 \wedge c.0 \wedge out.0]
\end{aligned}$$

Once again the stored charge on the internal nodes of N will have no effect on the steady state response of out, and hence the formulas for these node variables can be eliminated.

These formulas appear more complex than the previous ones, but in fact only require seven binary operations beyond the number required to represent the formulas for T and T_D. Hence, for the series-parallel case, the analyzer has a performance with α slightly more than 2. To better understand these formulas, consider the effect of a sequence in which the clock c is first set to 0 and then to 1. The first setting would give a steady state response $Out.1 = 1$ and $Out.0 = 0$. Letting these be the values of $out.1$ and $out.0$, respectively, the second setting would give a steady state response with $Out.1 = T_D(i_1.0,\ldots,i_k.0)$ and $Out.0 = T(i_1.1,\ldots,i_k.1)$. Hence, a domino gate has the same functionality as a static gate, even when some inputs equal X. The same result occurs when simulating the gate in ternary mode[31], with c set first to 0, then to X, and finally to 1. This indicates that the gate is not sensitive to the rise time of the clock.

In summary, the analyzer performs very well for most logic gates, deriving formulas that correspond directly to the series-parallel structure of the pulldown and pullup networks. Furthermore, even in subnetworks containing logic gates connected to more complex circuitry through pass transistors, the gate functions will be extracted efficiently. This performance should not seem extraordinary, given that logic gates form a particularly simple class of MOS circuits. However, this performance far exceeds the exponential complexity achieved by other general analysis methods. Having the general analysis algorithm

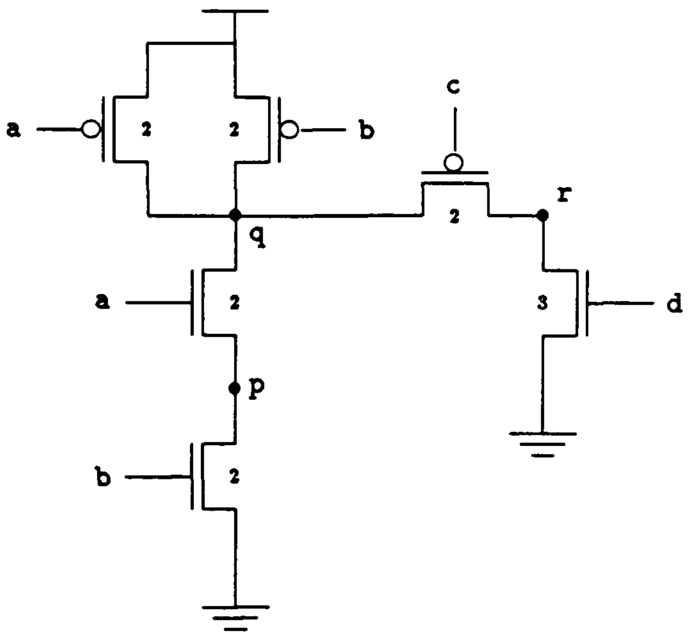


Figure 3: **Example CMOS Circuit.** Transistors are labeled by strength. All storage nodes have size 1. Although rather contrived, this circuit demonstrates a variety of interesting switch-level effects.

obtain efficient results for straightforward cases eliminates the need to devise specialized code to handle these cases.

6.2 Circuit with Stored Charge

Figure 3 shows a small, but relatively complex circuit in terms of its set of possible behaviors. This example demonstrates such properties as complementary logic, bidirectional pass transistors, stored charge, and ratio effects. It is not intended to demonstrate good circuit design practice. The circuit contains a two input NAND gate with output q. The gate output is connected by a p-type pass transistor to a node r which also has a "kill" transistor to ground. The kill transistor has strength 3, indicating that it can override any value transmitted to r through the pass transistor. Furthermore, if both the kill and pass transistors are turned on when the gate is driving toward 1, a fight will develop at q giving X as its steady state response.

The steps in the analysis are too complex to show in detail. Executing them and simplifying the formulas by hand yields the following results for node q:

$$\begin{aligned} Q.1 &= a.0 \vee b.0 \\ Q.0 &= a.1\,b.1 \vee c.0\,d.1. \end{aligned}$$

For clarity, the $\wedge$ symbols have been removed in the above formulas, and parentheses

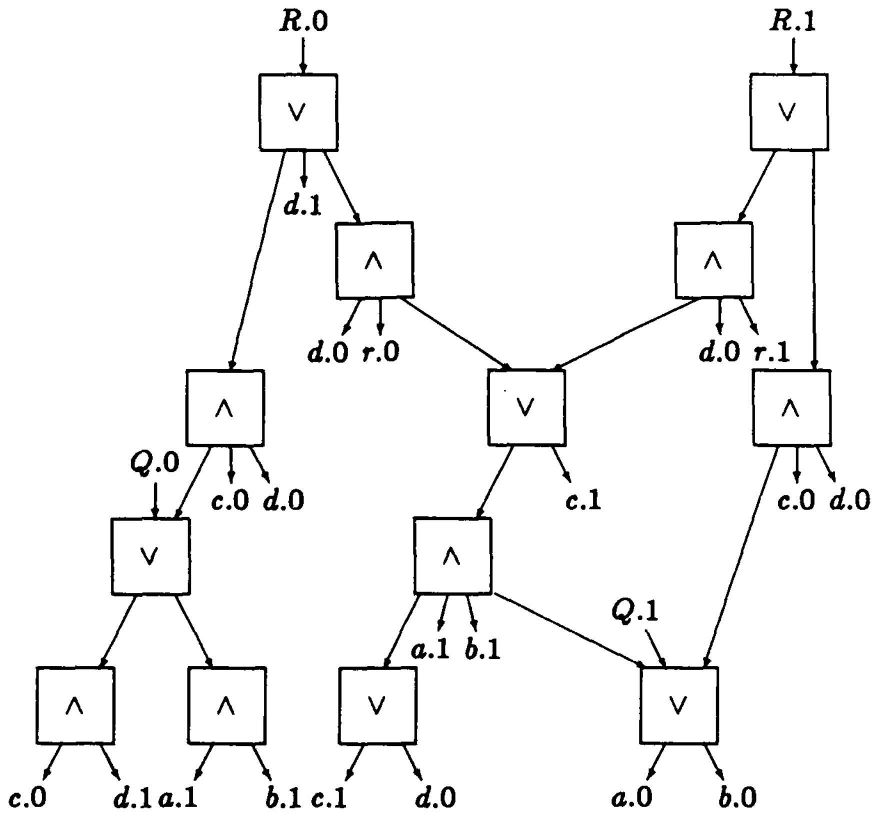


Figure 4: **DAG Representation of Example Circuit.** The leaves denote variables describing the initial state of the circuit, while the vertices denote Boolean operations. The pointers labeled $Q.1$, $Q.0$, $R.1$, and $R.0$ denote formulas for the new states of nodes q and r.

have been omitted where possible with the convention that AND takes precedence over OR. These formulas are simply those of a NAND gate with an additional term indicating the conditions under which the kill and pass transistor can drive q toward 0.

For r the analyzer yields the formulas:

$$\begin{aligned} R.1 &= d.0\left[c.1\,r.1 \lor (a.0 \lor b.0)c.0 \lor (a.0 \lor b.0)\,a.1\,b.1\,r.1\right] \\ R.0 &= d.1 \lor d.0\left[c.1\,r.0 \lor a.1\,b.1\,c.0 \lor (a.0 \lor b.0)\,a.1\,b.1\,r.0\right] \end{aligned}$$

These formulas appear quite complex, but each term can be recognized as describing a specific contribution to the steady state response. Part of this complexity owes to the fact that they describe the circuit behavior with some inputs equal to X as well as 0 and 1. From these formulas one can determine the conditions leading to the three possible steady state responses on r. The steady state will equal 0 for the following combinations of input and state variables, where a dash indicates that the corresponding variable can equal 0, 1, or X:

a	*b*	*c*	*d*	*r*
–	–	–	1	–
–	–	1	–	0
1	1	0	–	–
1	1	–	–	0

Similarly, it will equal 1 for the following combinations of input and state variables:

a	*b*	*c*	*d*	*r*
–	–	1	0	1
0	–	0	0	–
–	0	0	0	–
0	–	–	0	1
–	0	–	0	1

All other cases yield X.

Finally, the formulas for p are unimportant. Being an interconnect node within a logic gate, this node is not essential.

Figure 4 shows the DAG representation of the formulas for the steady state response on nodes q and r. This DAG has 13 nodes, representing 18 binary operations, giving an α of 3. Thus, even a more complex structure in terms of its switch-level behavior has a reasonably concise Boolean description. The size seems especially reasonable considering that the formulas describe the circuit outputs for all 729 ternary combinations of the 4 input and 2 state variables, not just the 64 Boolean combinations.

7 Extensions

Some applications of the switch-level model require features beyond the basic behavioral representation developed so far. Many of these extensions can be provided in straightforward way by modifying the Boolean equations for the steady state response. The ease with which these extensions are incorporated further demonstrates the strength of the mathematical framework.

7.1 Fault Modeling

Switch-level fault simulators such as FMOSSIM [3,4] have proved very successful at realistically modeling a wide range of faults in MOS circuits. These programs can represent the effects of faults such as nodes forced to ground or supply, as well as transistors stuck open or closed without changing the basic logic model. Normally, the analyzer produces formulas that cannot model these fault effects. Simply injecting faults into the formulas would create faults for which there are no counterparts in the switch-level network as well as overlook faults that could exist in the network. However, a modified analysis can preserve the fault behavior of the circuit. This modified analysis requires only a small number of additional algebraic operations (4 per transistor and 2 per node), although the resulting formulas cannot be simplified as well.

The analyzer injects faults by introducing additional Bool^an variables, where a variable is set to 1 when the fault is present and to 0 otherwise. This "fault variable" approach can describe fault effects by Boolean operations and hence apply the solution and manipulation techniques already developed.

For each node n, fault variable $n.\mathbf{o}$ indicates whether the node acts as an input or a storage node. As an input node, it is stuck at either 0, 1, or X depending on the values of $n.1$ and $n.0$. The analyzer incorporates this variable into the analysis by redefining the terms representing signals of input strength:

$$\begin{aligned} N.1_{w} &= n.\mathbf{o} \wedge n.1, \\ N.0_{w} &= n.\mathbf{o} \wedge n.0, \\ N.c_{w} &= \neg n.\mathbf{o}. \end{aligned}$$

The remainder of the analysis proceeds as before.

For each transistor t, variables $t.\mathbf{o}0$ and $t.\mathbf{o}1$ indicate the conditions when the transistor is stuck-open (nonconducting) or stuck-closed, respectively. Of course, a stuck-closed fault on a d-type transistor has no effect and hence can be omitted. When a transistor is stuck-closed, it is modeled at its nominal strength, although this can easily be generalized to different strengths. The analyzer incorporates these variables into the analysis by simply modifying the definitions for *indefinite*(t) and *potential*(t) as follows:

type	*indefinite*(t)	*potential*(t)
n-type	$(n.0 \wedge \neg t.\mathbf{o}1) \vee t.\mathbf{o}0$	$(n.1 \vee t.\mathbf{o}1) \wedge \neg t.\mathbf{o}0$
p-type	$(n.1 \wedge \neg t.\mathbf{o}1) \vee t.\mathbf{o}0$	$(n.0 \vee t.\mathbf{o}1) \wedge \neg t.\mathbf{o}0$
d-type	$t.\mathbf{o}0$	$\neg t.\mathbf{o}0$

All other steps of the analysis remain unchanged.

The analyzer can model other classes of faults, such as a bridges and breaks in the wires, by introducing additional fault variables. However, its efficiency degrades as the number of fault effects grows large, especially those effects that force a merging of subnetworks.

7.2 Restoring Logic

Conventionally, switch-level simulators ignore voltage degradations through pass transistors caused by threshold effects. This can cause a significant class of design errors to remain undetected. Many CMOS circuits, for example, are designed with the intention that only signals equal to either the supply or ground voltage act as valid logic values. A more conservative model would enforce this rule by yielding X when a 1 passes through an n-type or a 0 passes through a p-type transistor. In cases such as a transmission gate where each signal also passes through a complementary transistor, a 1 or 0 should result. The simulator MOSSIM II [34] optionally enforces such a rule. Yoeli and Brzozowski have proposed a similar rule in their switch-level model [35].

We can incorporate this convention into the switch-level model by modifying the definition of a definite path and consequently the way the analyzer computes $N.c_s$. That is, a path p with $Root(p) = \mathrm{n}$ is definite if either $n = 1$ (respectively, 0) and no transistor in $Trans(p)$ has state X or is of n-type (resp., p-type). In other words, only fully restored signals transmitted through fully conducting transistors can block weaker paths. Note that we need not be concerned about definite paths originating at a node with state X, because if such a path exists, and there is no definite stronger path, then the steady state response will equal X anyhow.

The computation of $N.c_s$ must compute the effects by sources of 1 and 0 separately and combine the two results:

$$N.c_s(n) \;=\; N.c1_s \wedge N.c0_s,$$

where formula $N.c1_s$ (respectively, $N.c0_s$) describes the absence of a definite path to node n originating at a node with state 1 (resp., 0) and having strength greater than or equal to s. the analyzer computes these two values by solving dual systems $[Indef1_s, initc1_s]^D$ and $[Indef0_s, initc0_s]^D$. To formulate these dual systems, define $indefinite1(t)$ and $indefinite0(t)$ for transistor t as follows:

type	$indefinite1(t)$	$indefinite0(t)$
n-type	$\mathbf{1}$	$n.0$
p-type	$n.1$	$\mathbf{1}$
d-type	$\mathbf{0}$	$\mathbf{0}$

That is, $indefinite1(t)$ (respectively, $indefinite0(t)$) describes the cases where the transistor cannot be part of a definite path originating at a node with state 1 (respectively, 0), where $\mathbf{1}$ indicates "always", and $\mathbf{0}$ indicates "never".

Edge labeling $\mathit{Indef1}_s$ is defined as $\mathit{Indef1}_w(\mathrm{m},\mathrm{n}) = \mathbf{1}$ and for $w > s > 1$ as

$$\mathit{Indef1}_s(\mathrm{m},\mathrm{n}) = \mathit{Indef1}_{s+1}(\mathrm{m},\mathrm{n}) \wedge \bigwedge_{t \in \mathcal{T}_s(\mathrm{m},\mathrm{n})} \mathit{indefinite1}(t).$$

Vertex labeling $\mathbf{\mathit{initc1}_s}$ is defined for storage node n and strength $w > s > 1$ as

$$\mathit{initc1}_s(\mathrm{n}) = \begin{cases} N.c1_{s+1} \wedge \bigwedge_{\mathrm{m} \in \mathcal{N}_w} \bigwedge_{t \in \mathcal{T}_s(\mathrm{m},\mathrm{n})} [\mathit{indefinite1}(t) \vee m.0] & s > k \\ n.0 & \mathrm{n} \in \mathcal{N}_s \\ N.c1_{s+1} & \text{else} \end{cases}$$

with the convention that $N.c1_w = \mathbf{1}$. Observe how this formula uses variables $n.0$ and $m.0$ to place restrictions on the root node state. If $n.0 = 1$, then node n cannot be the root of a definite path p with state 1, and similarly for $m.0$.

Similarly, edge labeling $\mathit{Indef0}_s$ is defined as $\mathit{Indef0}_w(\mathrm{m},\mathrm{n}) = \mathbf{1}$, and for $w > s > 1$ as

$$\mathit{Indef0}_s(\mathrm{m},\mathrm{n}) = \mathit{Indef0}_{s+1}(\mathrm{m},\mathrm{n}) \wedge \bigwedge_{t \in \mathcal{T}_s(\mathrm{m},\mathrm{n})} \mathit{indefinite0}(t).$$

Vertex labeling $\mathit{initc0}_s$ is defined as

$$\mathit{initc0}_s(\mathrm{n}) = \begin{cases} N.c0_{s+1} \wedge \bigwedge_{\mathrm{m} \in \mathcal{N}_w} \bigwedge_{t \in \mathcal{T}_s(\mathrm{m},\mathrm{n})} [\mathit{indefinite0}(t) \vee m.1] & s > k \\ n.1 & \mathrm{n} \in \mathcal{N}_s \\ N.c0_{s+1} & \text{else} \end{cases}$$

with the convention that $N.c0_w = \mathbf{1}$.

This extension adds an extra system of equations at each level and hence results in somewhat larger formulas. This seems a reasonable price to pay for detecting an additional class of circuit design errors.

As an example, applying this modified analysis to the circuit of Figure 3 has no effect on node q, but for node r yields the formulas:

$$\begin{aligned} R.1 &= d.0\left[c.1\,r.1 \vee (a.0 \vee b.0)c.0 \vee a.1\,b.1\,r.1\right] \\ R.0 &= d.1 \vee d.0\left[c.1\,r.0 \vee a.1\,b.1\,c.0 \vee a.1\,b.1\,r.0\right] \end{aligned}$$

This reduces the cases for which r has steady state response 0 to the following:

a	b	c	d	r
–	–	–	1	–
–	–	1	–	0
1	1	–	–	0

That is, r will not be pulled to 0 when $a = b = 1$ and $c = 0$, unless $r = 0$ or $d = 1$.

7.3 Charge Decay

During normal operation, most switch-level simulators assume that a node retains its stored charge indefinitely. In actual circuits leakage currents cause stored charge to eventually decay to some indeterminate value. A simulator that does not model this decay will fail to detect cases in which proper behavior depends on stored charge being maintained beyond some reasonable time limit.

The simulator MOSSIM II has an optional mode in which charge is retained only for a number of clock cycles specified by the user. Any storage node that remains unrefreshed for this many cycles is set to X. Such an event does not in itself indicate a circuit design error. However, any further operations that depend on this node state will yield more X values, and hence invalid uses of stored charge can be detected. MOSSIM II implements this feature by tagging every node with the most recent refresh time, measured in clock cycles. Due to subtleties caused by both charge sharing and by transistors in the X state, it must use a rather complex algorithm to compute the effective refresh time of a node as it updates the state.

The capability provided by MOSSIM II cannot be described efficiently in terms of Boolean operations, because every node state must specify both a logic value and an integer refresh time. By adopting a somewhat less precise timing scheme, however, a Boolean representation becomes more practical. In this scheme, the period over which the circuit operates is divided into a series of "epochs". An epoch will typically have duration equal to half the maximum number of clock cycles for which stored charge may be assumed valid. A flag is maintained for each storage node indicating whether the node is "fresh" or "stale", i.e., whether or not it has been refreshed during the current epoch. This flag is updated every time the node state is recomputed. At the end of each epoch the states of all stale nodes are set to X. At the same time, all nodes not currently connected to input nodes are marked as stale for the start of the next epoch. With this scheme, the exact charge retention time can range between just over one epoch to just under two, depending on the alignment between the refresh time and epoch boundaries. This degree of accuracy suffices for most applications, because most designers set conservative limits on charge retention time.

Developing a precise definition of the conditions under which a node is refreshed involves several subtleties. Clearly, a node is refreshed whenever it is connected to an input node by a set of transistors in the 1 state. Consider, however, the circuit shown in Figure 5 in which several storage nodes may share charge. In particular, node p has greater size than either q or r, and hence the state of p can override those of q and r. In some circuit designs, such as where p is high capacitance bus, the transistors may be operated in such a way that no conducting path ever forms between an input node and either q or r. Such cases can be handled with a convention that whenever nodes q or r share charge with p, they will be marked as fresh if p is fresh and as stale otherwise. As a further subtlety, when transistors in the X state are present, a node may or may not be refreshed depending whether or not this transistor is actually conducting. Such cases can be handled with a convention that a node should be marked as stale if its state may depend on that of some stale node

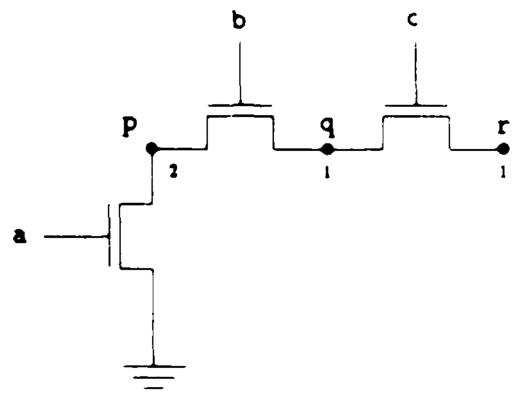


Figure 5 **Charge Sharing Circuit Example.** Nodes q and r can obtain new logic values by sharing charge with p.

for some combination of conducting and nonconducting transistors. To summarize these conventions precisely, a node should be marked as stale whenever it is the destination of an unblocked path originating at a stale node.

To express these conditions symbolically, the analyzer introduces a variable $n*$ for every storage node n, with value 1 indicating "stale" and 0 indicating "fresh". The analyzer then generates a formula $\mathcal{N}*$ for each node specifying when it should be marked as stale as a function of the state and refresh variables associated with the nodes. The analyzer generates these formulas in a manner similar to that used to generate the formulas $\mathcal{N}1$ and $\mathcal{N}0$. Starting at strength $s = k$ (the maximum storage node size), and working downward to 1, it solves the system $Conduct_s, init*_s$ to generate formulas $\mathcal{N}*_s$ for each node n. The desired formula $\mathcal{N}*$ equals $\mathcal{N}*_1$. The edge labeling $Conduct_s$ has already been defined as Equation 10. The vertex labeling $init*_s$ indicates the conditions under which each node may be the source of stale charge.

$$init*_s(n) = \begin{cases} \mathcal{N}*_{s+1} + \mathcal{N}c_{s+1} \cdot n* & n \in N_s \\ \mathcal{N}*_{s+1} & \text{else.} \end{cases}$$

with the convention that $\mathcal{N}*_{k+1} = 0$.

The simulator utilizes these formulas as follows. At the end of an epoch, the simulator makes two passes over the nodes. The first pass sets the state of any node n for which $n* = 1$ to X and also sets $n*$ to 1. The second pass evaluates the formula $\mathcal{N}*$ to determine the new value of $n*$. This second pass marks as fresh only nodes to which all unblocked paths originate at input nodes. As the simulator proceeds, every time it computes a new state for node n, it computes a new value of $n*$ by evaluating the formula $\mathcal{N}*$.

As an example, applying this analysis to the circuit of Figure 5 yields the following formulas:

$$\begin{aligned} P.* &= a.0\,p.* \\ Q.* &= a.0\,b.1\,p.* \vee b.0\,q.* \vee b.0\,c.1\,r.* \\ R.* &= a.0\,b.1\,c.1\,p.* \vee b.0\,c.1\,q.* \vee b.0\,r.* \vee c.0\,r.*. \end{aligned}$$

Observe that p can be marked as stale only if it is already stale and $a \neq 1$. On the other hand, q is marked as stale if it shares charge with a stale value on either p or r, or if it is isolated and already stale. Similar results hold for r.

8 Conclusion

Transforming a switch-level network into an explicit functional representation has proved a challenging task. Previous attempts yielded results that were too inefficient or too inaccurate for practical use. The solution presented here relies on three major ideas. First, systems of Boolean equations can describe switch-level networks. Second, Gaussian elimination can take advantage of the sparse structure of these systems and generally give solutions of linear complexity. Finally, the DAG representation of a set of formulas can exploit the sharing of common subexpressions to give a very compact result.

The analyzer has the potential to improve the efficiency of programs for a variety of MOS circuit analysis tasks. It can incorporate a number of modeling extensions by modifying or augmenting the system equations. The advantage of this approach to switch-level modeling will increase as hardware becomes available that achieves high performance through greater degrees of specialization and concurrency.

References

1 R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers* Vol. C-33, No. 2 (February, 1984) pp. 160–177.

2 C. J. Terman, *Simulation Tools for Digital LSI Design*, PhD Thesis, MIT Dept. Elec. Eng. and Comp. Sci. (October, 1983).

3 M. D. Schuster, and R. E. Bryant, "Concurrent Fault Simulation of MOS Digital Circuits", *Advanced Research in VLSI*, P. Penfield, Jr., *ed.*, MIT (1984), pp. 160–177.

4 R. E. Bryant, and M. D. Schuster, "Performance Evaluation of FMOSSIM, a Concurrent Switch-Level Fault Simulator", *22nd Design Automation Conf.*, ACM (1985), pp. 715–719.

5 H. H. Chen, R. G. Mathews, and J. A. Newkirk, "An Algorithm to Generate Tests for MOS Circuits at the Switch-Level", *International Test Conf.*, IEEE (1985).

6 M. K. Reddy, S. M. Reddy, and P. Agrawal, "Transistor Level Test Generation for MOS Circuits", *22nd Design Automation Conf.*, ACM (1985) pp. 825–828.

[7] R. E. Bryant, "Symbolic Verification of MOS Circuits," *1985 Chapel Hill Conf. on VLSI,* H. Fuchs, *ed.* Computer Science Press (1985), pp. 419–438.

[8] D. S. Reeves, and M. J. Irwin, "Functional Verification of Digital MOS Circuits", *IEEE International Conf. on Computer-Aided Design*, (1986), pp. 496–499.

[9] E. Ulrich, and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks", *IEEE Computer* (April, 1974), pp. 39–44.

[10] M. M. Denneau, "The Yorktown Simulation Engine", *19th Design Automation Conf.*, ACM (1982), pp. 55–59.

[11] *Daisy Megalogician Product Description*, Daisy Systems, 1984.

[12] *ZyCad LE-001 and LE-002 Product Description*, ZyCad Corp., 1982.

[13] W. J. Dally and R. E. Bryant, "A Hardware Architecture for Switch-Level Simulation", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-4, No. 3 (July, 1985), pp. 239–249.

[14] E. H. Frank, "Switch-Level Simulation of VLSI Using a Special-Purpose, Data-Driven Computer", *22nd Design Automation Conf.*, ACM (1985) pp. 735–738.

[15] G. Pfister, private communication, 1980.

[16] Z. Barzilai, *et al*, "Simulating Pass Transistor Circuits Using Logic Simulation Machines", *19th Design Automation Conf.*, ACM (1983), pp. 157–163.

[17] J. Hayes, "A Unified Switching Theory with Applications to VLSI Design", *Proc. IEEE*, Vol. 70, No. 10 (October, 1982), pp. 1140–1151.

[18] Z. Barzilai, *et al*, "SLS—a Fast Switch Level Simulator for Verification and Fault Coverage Analysis", *23rd Design Automation Conf.*, ACM (1986), pp. 164–170.

[19] E. Cerny, and J. Gecsei, "Simulation of MOS Circuits by Decision Diagrams", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, Vol. CAD-4, No. 4 (October, 1985), pp. 685–693.

[20] G. Ditlow, W. Donath, and A. Ruehli, "Logic Equations for MOSFET Circuits", *International Symposium on Circuits and Systems*, IEEE (1983), pp. 752–755.

[21] I. N. Hajj, and D. Saab, "Symbolic Logic Simulation of MOS Circuits", *International Symposium on Circuits and Systems*, IEEE (1983).

[22] C. A. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[23] M. R. Garey, and D. S. Johnson, *Computers and Intractability*, Freeman, 1979.

[24] L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, PhD Thesis, Univ. of California, Berkeley, Dept. of Elec. Eng., 1975.

[25] R. E. Bryant, *Algorithmic Aspects of Symbolic Switch Network Analysis*, companion paper, 1987.

[26] C. Lutz, S. Rabin, C. Seitz, and D. Speck, "Design of the MOSAIC Element," *Advanced Research in VLSI*, P. Penfield, Jr., *ed.*, MIT (1984), pp. 1–10.

[27] R. Byrd, G. D. Hachtel, and M. R. Lightner, *Switch Level Simulation: Part I—Theory and Algorithmic Frame*, unpublished, 1985.

[28] M. Yoeli, and S. Rinon, "Application of Ternary Algebra to the Study of Static Hazards," *J.ACM*, Vol. 11, No. 1 (January, 1964), pp. 84–97.

[29] L. A. Glasser, and D. W. Dobberpuhl, *The Design and Analysis of VLSI Circuits*, Addison-Wesley, 1985.

[30] N. H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, 1985.

[31] R. E. Bryant, "Race Detection in MOS Circuits by Ternary Simulation", *VLSI83*, F. Anceau and E. J. Aas, *ed.* North-Holland (1983), pp. 85-95.

[32] M. Takashima, *et al*, "Programs for Verifying Circuit Connectivity of MOS/LSI Mask Artwork", *19nd Design Automation Conf.*, ACM (1982), pp. 544–550.

[33] C. Ebeling, and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison", *IEEE International Conf. on Computer-Aided Design*, (1982), pp. 172–173.

[34] R. E. Bryant, M. D. Schuster, and D. Whiting, *MOSSIM II: A Switch-Level Simulator for MOS LSI, User's Manual*, Technical Report 5033, Dept. of Comp. Sci., Caltech (1982).

[35] M. Yoeli, and J. A. Brzozowski, "A Mathematical Model of Digital CMOS Networks", *Canadian Conf. on VLSI* (1985).

END
DATE
FILMD
3-88
DTIC